A COMPREHENSIVE QUERYING DATABASE SYSTEM BASED ON THE ENTITY RELATIONSHIP MODEL

A Thesis
Presented to
The Academic Faculty

by

Moritz Wilfer

In Partial Fulfillment
of the Requirements for the Degree
Master of Computer Science in the
College of Computing

Georgia Institute of Technology
May 2015

A COMPREHENSIVE QUERYING DATABASE SYSTEM BASED ON THE ENTITY RELATIONSHIP MODEL

Approved by:

Professor Shamkant B. Navathe, Advisor
College of Computing
Georgia Institute of Technology

Professor Edward R. Omiecinski
College of Computing
Georgia Institute of Technology

Professor Mayur Naik
College of Computing
Georgia Institute of Technology

Date Approved: 20 April 2015

To my brother,

my parents,

and my friends,

who always believed in me and supported me.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Prof. Shamkant Navathe for his continuous support. His expertise in the field of databases helped me to reach my full potential.

Besides my advisor, I would like to thank my co-advisors from TU Munich, Prof. Thomas Neumann, Prof. Alfons Kemper, and Viktor Leis. Their outstanding research in the area of high performance databases is essential for this work.

My sincere thanks also goes to the person responsible for the double degree program at TU Munich, Prof. Michael Gerndt.

Last but not the least, I would like to thank my family and friends. They made me the person I am and have always supported me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

This work proposes a comprehensive querying database system based on an *enhanced entity relationship (EER)* model. The DBMS is fully operational and performs all queries that are illustrated in the paper. This work is also applicable for the general ER model proposed by Chen [6]. So far, the ER model is mainly used by database designers as a conceptual model during the database design phase. An ER schema is usually mapped into a representation in the logical model of the targeted database. As an analogy, the ER schema represents a program, written in a higher level language that is compiled into a lower level machine-executable equivalent. Semantics like the relationships among entities or the cardinality ratio constraints may no longer be available at the logical model level. Queries are then written against the logical model, which generates a discrepancy between the view of the database designer and the view of the database user. This work bridges this gap by introducing an EER-algebra and a high-level query language called *ERSQL*. The algebra is heavily based on the general ER-algebra proposed by Parent and Spaccapietra [23, 24]. To provide a semantic foundation for ERSQL we introduce a canonical translation algorithm that maps an ERSQL query into an EER-algebra expression. In recent years, in NoSQL data stores the functional primitives are greatly simplified for performance and scalability reasons. Our ERDBMS goes in the opposite direction: We use CISC (complex instruction set) operators but implement them efficiently in main-memory data storage.

CHAPTER I

INTRODUCTION

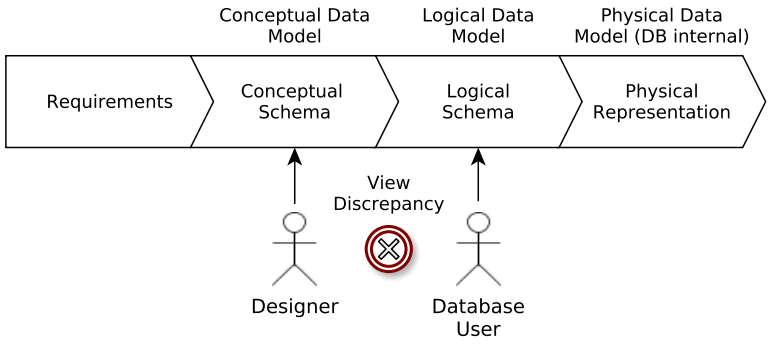


Figure 1: Standard four step database design process

When Chen proposed his ER model [6], he also introduced mapping (derivation) techniques to the relational, network, and entity-set model as a set of logical data models. We will use the term logical data model to represent a model that is used to map data into storage and then allows the user to form queries and write transactions using a DBMS that supports that model. Thus, a data model qualifies as a logical model if a sound basis for querying exists. While Chen presented these mapping techniques to show the power of his ER model, they gained importance over time. The mapping technique into the relational model is used especially during the design phase of almost every relational database. Storey [26] describes a widely accepted four step approach to designing a database (see also Fig. 1):

1. *Requirement specification*: All informational needs of the mini world to be modeled are identified.

2. *Conceptual design*: The user and application view of the mini world is modelled. The requirements of Step 1 are taken into account, and a higher level representation of the requirements is generated. An example for a model that is capable of expressing such a higher level conceptual representation is the ER model.

3. *Logical design*: The conceptual schema is translated into a logical schema that

corresponds to the logical data model of the targeted *database management system (DBMS)*.

4. *Physical design*: This step is usually internal to the targeted DBMS and maps the logical schema into a physical representation on the actual hardware.

Due to the lack of a sound and standardized foundation for querying, the ER model did not get qualified as a logical data model so far. There have been many approaches to provide an algebraic [5, 18, 21] or a calculus [11, 14] foundation for the ER model but none of them have become widely accepted. They either only support a subset of the ER model or do not incorporate the concepts of the ER model in a satisfactory manner. Omodeo [21] tries to use an adapted relational algebra and a model translation technique. However, we consider the concepts of relations and entities too different in their nature. Relations are flat, while entities can have multivalued attributes. The ER model with its easy but extensive modelling facilities represents a perfect conceptual model. One major disadvantage of the mentioned four-step database design approach is the discrepancy between the designers' view and the database users' view. While the designer talks about entities of the mini world and the relationships among them, a database user writes queries against the representation of them in the logical data model. The user has a lower level and less abstract picture of the mini world which can be a disadvantage. An entity (e.g. a student or department in a university database schema) is often mapped into multiple relations, which makes it harder to query them. A simple multivalued attribute for an entity in the ER schema leads to an additional relation in the relational model. Composite attributes cannot be represented as such by the relational model. Relationships among entities are also mapped into simple relations which leads to a loss of semantics. Foreign key constructs cannot offer the same meaning as relationship

constructs in the ER model. These examples only scratch the surface of the discrepancies between the designers' view and the database user's view. To overcome this issue, some have proposed multiple high-level query languages to transform the ER model into a logical data model [9, 16, 17]. A major problem with these proposals is the lack of a semantic foundation. The query languages are either introduced in an exemplary fashion or with a rather poor descriptive semantic foundation. We view this as a major drawback. Furthermore, some of them show navigational characters, which makes query formulation harder. We consider a declarative query language important for user acceptance. Uwe Hohenstein and Gregor Engels' SQL/EER [14] language proposal on the other hand provides a sound semantic foundation by means of a mathematical representation of the used EER model and a translation technique of the SQL/EER queries in a well-defined EER calculus [11]. However, the problem with their approach is twofold. First, they do not incorporate the concept of multivalued attributes in a satisfactory fashion, thus, the results of their calculus expressions are unnecessarily flat. Furthermore, the results are no longer covered by concepts of their targeted EER model. They are neither valid entities, nor relationships, nor attributes and thus cannot be used as operands in other calculus expressions. Parent et al. [22, 23, 24] faced these issues by proposing an algebra and an equivalent calculus for an EER model that generates valid entities as a result of an algebra or calculus expression. Moreover, they abandoned the flatness of the relational model by incorporating multivalued attributes in a satisfactory way. We consider their proposals very promising and our work is heavily based on their work. We will present a slightly adapted and extended algebra in detail in Section 4. While Hohenstein and Engels claim that aggregation and aggregation functions such as *count*, *average*, *max*, and *min* must be supported by the underlying calculus, our algebra achieves aggregation for free by incorporating aggregation into multivalued attributes. We provide more details on this in Section 3. All the mentioned proposals shared one major issue:

due to the complexity of the ER model, the proposed functional primitives are also more complex than their relational counterparts. It was hard or even impossible to implement them in an efficient fashion. Now, modern implementation techniques and increased computational power make CISC operators efficiently implementable. Our proposal goes in the opposite direction of NoSQL stores which simplify functional primitives for scalability and performance reasons. We use complex functions but implement them efficiently. We provide more details on this in Section 6. Moreover, we believe that the success of the relational model is closely related to the power and convenience of SQL, thus, we decided that our ERSQL query language should be declarative and geared to SQL. This approach was also followed in SQL/EER [14] and partially in the SERQL language proposed by Gene T. J. Wuu [28].

A further goal of our work is to provide a proof-of-concept implementation of the querying facilities. The implementation should show that the theoretical constructs discussed in this work are implementable. Therefore, we developed a comprehensive ERDBMS prototype which implements our algebra and ERSQL.

The remainder of this work is organized as follows. Section 2 provides the motivational aspects for the development of an EER algebra and a high-level query language. Section 3 briefly introduces the EER model on which we based our algebra and query language. Section 4 provides a detailed introduction of our EER algebra, which provides the semantic foundation for our ERSQL query language. Section 5 introduces ERSQL and the canonical translation algorithm. Section 6 introduces our prototype ERDBMS. Finally, section 6 presents related work and our conclusions. The queries illustrated in this paper can be expressed in ERSQL, which is already developed.

CHAPTER II

MOTIVATION

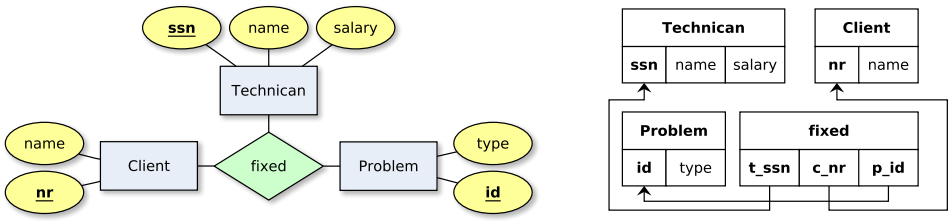


Figure 2: A ternary relationship in an ER schema (left) and the relational representation (right)

In this section, we present several reasons for the development of querying facilities for the ER model. We will use the relational model as an example to show the shortcomings of the querying facilities in a current logical data model.

2.1 Bridging the semantic gap

Entities and their interrelationships are modelling constructs that are close to the natural perception of human beings. Treating the ER model as a logical data model bridges the semantic gap in the views of the database designer and the database user. Database users can write their queries against the same data model and schema that the database designer works on. As a consequence, the user can work with the real mini world entities and their relationships among each other instead of dealing with the lower level representation of them in the relational model. The database user would work directly with an abstract but natural view of the mini world which simplifies query formulation.

2.2 Relational Algebra is RISC, ER-Algebra is CISC

Reduced instruction set computing (RISC) and complex instruction set computing (CISC) are common to the processor architecture domain. They fit perfectly when we compare the functional primitives of an ERDBMS with those of a traditional DBMS. A query in relational algebra needs less operators in the EER algebra. This is a consequence of the fact that the ER algebra operators are richer and more complex.

Fig. 2 illustrates a ternary relationship *fixed* which links a technican, a client, and a problem. The right side shows the relational representation of the same schema. If we want to have the clients of the technican Doe with the problem he has fixed for them, several joins are needed in the relational algebra: $((\sigma_{name='Doe'}(Technican) \bowtie_{ssn=t\_ssn} fixed) \bowtie_{c\_nr=nr} Client) \bowtie_{p\_id=id} Problem$. Using the ER algebra, we only need two operators: $\sigma_{name='Doe'}(Technican) \bowtie_{fixed} (Client, Problem)$. In Section 6, we will provide details on how to implement such CISC operators, namely, the presented relationship join and merge join.

2.3 A more natural query language

While SQL is overall considered a simple and easy to learn query language, there is still room for improvement. Joining two relations means relating pairwise attributes of both relations in a predicate in the *where* clause. Multiple joins can lead to an extensive and confusing *where* clause. While the ER schema provides evidence of how the entities may be joined by the user through relationships among them, this information is not readily available in the mapped relational schema. Thus, the user has to identify the attributes to join relations on his own. In Section 5, we show that exposing entities and the relationship among to the database user, ERSQL query formulation is easier. The user no longer has to deal with attributes to join two entities but only has to know the relationships between them. This leads to queries which are closer to natural language.

2.4 Extensive database constraints

The relational model offers key constraints, entity integrity constraints, and referential integrity constraints. The key constraints enforce a group of attributes of a relation to uniquely identify a tuple in the relation. The entity integrity constraints guarantee that key attributes cannot have null values. The referential integrity constraints guarantee consistency among distinct relations that are connected via some

relationship. A referencing relation R_1 can point to a referenced relation R_2 by including the primary key of R_1 as a foreign key. The referential integrity constraints guarantee that the value of the foreign key is either an existing primary key value in R_1 or is null. The ER model offers far more advanced constraints to control the database state. One type of constraints are the so called relationship constraints. These make it possible to control the number of relationships in which an entity can participate. There are two common notations that are used to specify such constraints. The first one is the *maximum cardinality ratio notation* in which one can specify a relation to be *1-to-1*, *1-to-N*, or *N-to-M*. The *minimum cardinality ratio* constraint indicates whether a relationship participation is optional or mandatory. A mandatory participation constraint ensures that each entity has to participate in at least one relationship. The alternative relationship constraint notation is the *min-max notation*. It is far more accurate as one can specify the minimal and maximal occurrences of an entity for each relationship set. In this work, we will focus on the min-max notation. We further investigate the question of whether it is possible to enforce min-max constraints with the functionality provided by SQL for a relational DBMS. Listing 2.1 shows how one can enforce min-max constraints.

Listing 2.1: Min-max constraint simulation with SQL

```
a)
begin transaction;
insert into works_on set employee_ssn=123456, project_nr=123;
select count(*) as current from works_on where employee_ssn=123456;
if current > MAX_EMP_WORK_ON then abort else commit;
b)
begin transaction;
delete from works_on where employee_ssn=123456 and project_nr=123;
select count(*) as current from works_on where employee_ssn=123456;
if current < MIN_EMP_WORK_ON then abort else commit;
```

Assume two such insert transactions (a) running concurrently. They are adding the same employee to two new projects. Further assume that the given employee

currently already works on 9 projects and that an employee can work on, at most, 10 projects. If we assume the highest transaction isolation level (We assume even *LOCKING SERIALIZABLE* isolation. According to the work of Berenson et al. [3] the ANSI isolation level formulations are ambiguous), the min-max constraints can be enforced successfully. However, to the best of our understanding, even the slightest reduction of the isolation level will lead to a violation of the max constraint. Berenson et al. have shown that the ANSI definition of phantom reads is ambiguous and thus situations like the one mentioned above will fail even if we avoid phantom reads in its strict interpretation. We are required to avoid phantom reads in its broadest interpretation. For more information on this topic, we refer to [3]. Another type of constraints that is available in the ER model are value set constraints. These help to control the format of the values that occur for certain attributes. We will provide more details on these constraints in Section 3.

2.5 Redundancy in the flat relational model

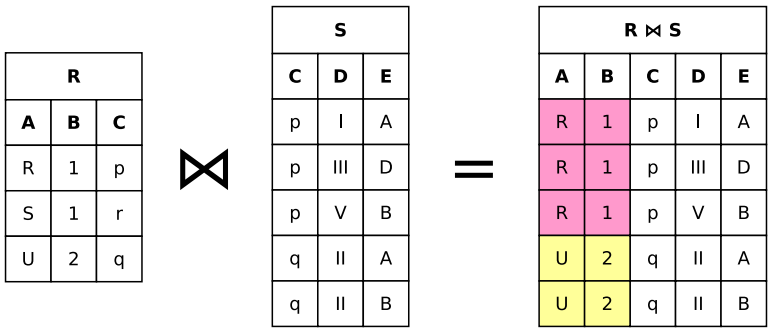


Figure 3: Redundancy in the result of a natural join between R and S (light yellow and purple)

Normal forms were introduced for the relational model to avoid redundancy and guarantee consistency in the data. But results of queries with joins can have a built-in redundancy due to the flatness of the relational model. Imagine that one joins

a relation R_1 with another relation R_2 and that a certain tuple t in R_1 has k join partners in R_2. Thus, the attributes of t occur k times in the result set (in combination with each join partner). This is also the reason why we decided not to use the EER calculus used by SQL/EER, because it treats joins very similarly to the way relational calculus treats them. It creates the same redundancy when joining two entities over a relationship. DBMS instances often run on remote servers that are connected to the application servers over a network. The redundancy leads to network bandwidth consumption. As DBMSs have become more and more efficient over time, the network often became the bottleneck in query processing. Thus, it is necessary to keep the data volume as small as possible. Parent and Spaccapietra [23, 24] faced this issue in their work by incorporating multivalued attributes and joins. This is why the algebra we will present in section 4 is heavily based on their work.

2.6 Arbitrary hierarchical export format

Data processing tools often lack a database import function and require the input data in a certain format, in *XML* or *JSON* for example. *SQL/XML* is an SQL extension and allows the generation of XML files directly from the database [7]. While the extension allows convenient XML generation, it still suffers from the flatness of the relational model. Hierarchical XML files can be generated only by subqueries in the *select* clause. The ER model allows ERSQL to generate entity types with arbitrary levels of multivalued attributes. With the help of a JSON print operator, we were able to generate JSON files directly with ERSQL. Adding XML extensions similar to those from SQL/XML will allow ERSQL to generate XML files without the need for any subquery.

2.7 Combining aggregation function with data retrieval

SQL offers aggregation functions such as *count*, *average*, *min*, and *max*. These functions collapse the result set into a single tuple, and the aggregated data itself is no

longer available in the results but only the values of the invoked aggregation function. Often, aggregate functions are used in combination with grouping on single or multiple attributes. The result set will contain a single tuple for each group. Each such tuple consists only of the values of the grouping attributes and the results of the invoked aggregation function(s).

A query like *"Find all employees of the computer-science department and the average salary of these employees"* shows a huge drawback of the way SQL implements aggregation and grouping because we need two queries to answer it. With the EER model and algebra we propose in this work and the way we implement aggregation functions (as defined for multivalued attributes), we can retrieve both, the value of such aggregation functions and the actual entities that fall into a certain group, within a single query.

2.8 Star-join queries

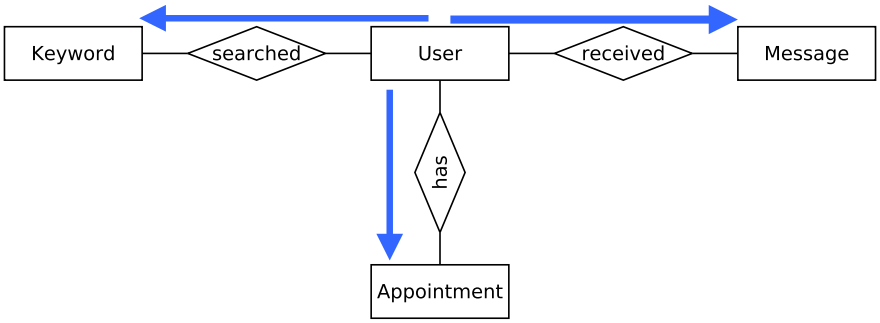


Figure 4: ER schema with an illustration of a comprehensive star join

This problem can be most easily illustrated by a web 2.0 user-tailored page, the page a user sees after login. Imagine such a page shows the appointments of a user, the messages he got, and the keywords he searched for in the past. Fig. 4 shows the corresponding ER schema and illustrates why we call such queries *star joins*. We join multiple entity sets from a central entity set.

In SQL, one would issue 3 separate SQL queries with a single join per query. However, this solution has a drawback: the query processing has to be invoked for each query. If the DBMS server is a remote machine connected via a network, an additional network roundtrip is added per query. The algebra and query language we propose in this work allows us to formulate such star-join queries without any overhead in the result size. This is an advantage in that the ER data model is not flat.

2.9 Simplifying natural language processing

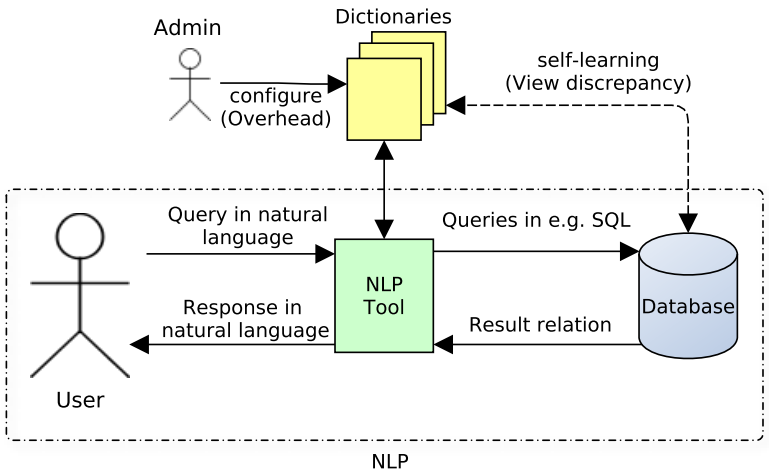


Figure 5: Natural language processing by the help of admin-configured dictionaries

There are already a large number of tools for *natural language processing (NLP)* in the context of database systems [1, 2, 13, 27]. Usually, they translate a natural language expression into a query for a given database system. This approach is illustrated in Fig. 5. The problem with these systems is that they either require an enormous amount of configuration or have rather limited language processing abilities (e.g. single relation queries only). We think that our system can drastically reduce the amount of configuration. For the MASQUE system [1, 2], the database administrator is required to specify a semantic dictionary, and a type hierarchy for a given

database before it is able to produce useful responses to queries expressed in natural language. This information is required by the system to classify the database into entities and their relationship amongst each other. For an ER-based database, this step should be no longer necessary or at least it should be reduced greatly because the database is already structured in the form of entities and because the relationships among them are well known. The ROBOT system [13], on the other hand, requires an extensive configuration effort to be robust against informal illnesses in the natural language. While the user acceptance evaluation results are quite promising, the ROBOT system has one drawback to the best of our understanding. It does not take into account relationships among different relations. Moreover, natural language querying interfaces often use the current database state for self-configuration. They try, for example, to infer existing entities in the modeled mini world by the names of relations in the database. As mentioned, the mapping process from an ER schema into a relational schema can cause one entity to be split into several relations. While phrases in natural languages often use the more abstract entities of the ER schema, the NLP tools work on the lower level mapped relation names. It is obvious that this leads to a gap between the users' perception of the mini world and the NLP tools' perception. Treating the ER model as a logical data model would bridge this perception gap.

CHAPTER III

THE TARGETED EER MODEL

The EER model, the target of our algebra and ERSQL query language, is based on the original ER model [6]. Our algebra and ERSQL are also applicable if just the original ER model is used. We decided to use an *enhanced ER (EER)* model to provide the user with facilities to model specializations of entity types. We follow the EER model presented in the book *"Fundamentals of Database Systems"* by Elmasri and Navathe [8]. The remainder of this section is threefold: First, we briefly present the important concepts of the original ER model presented by Chen. Second, we state our enhancements to the original ER model. Third, we show how we incorporated aggregation functions such as *count*, *average*, *min*, and *max* into the concept of multivalued attributes.

3.1 The general ER model by Chen

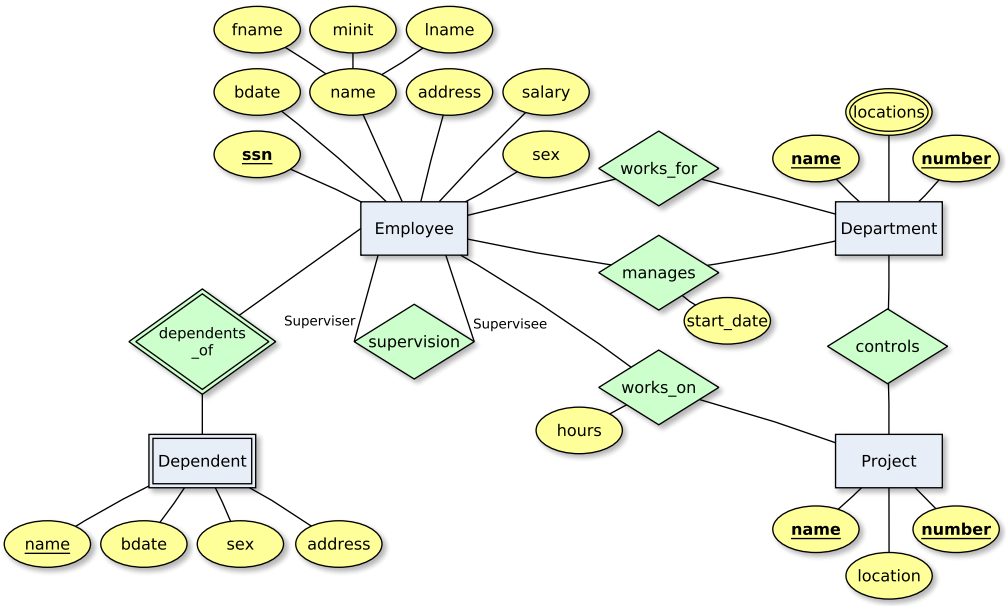


Figure 6: Exemplary ER schema based on original ER model (taken from [8])

The main concepts in the original ER model are entities, and their interrelationships. Chen originally described an entity as a thing in the mini world and a relationship as an association among entities. Each entity type and relationship type

can be equipped with an arbitrary number of attributes. An *Employee* entity type, for example, could have the attributes *name*, *age*, and *salary*. A *managed_by* relationship type between an *Employee* and a *Department* entity type could have an attribute *start_date*, which indicates when an employee became the manager of a department. An entity set is a collection of entities from the same entity type and a relationship set is a collection of relationship instances from the same relationship type. We will discuss the original ER model in a more formal way and present the graphical representation for an ER schema now.

3.1.1 Entities, entity sets, and entity types

An entity describes a uniquely identifiable thing in the mini world. Entity types are used to give an entity set of comparable entities a meaningful name. In a graphical representation of an ER schema, entity types are shown as rectangles. Fig. 6 includes the entity types *Employee*, *Department*, and *Project*. A special type of entities are so called weak entities. The existence of a weak entity is coupled with the existence of one or multiple strong entities. The interrelationships between them are called identifying relationships. An example for a weak entity type is a *Room* entity type because the existence of *Room* entities is depending on the existence of a *Building* entity. Weak entity types are graphically represented by double-lined rectangles. In Fig. 6, *Dependent* represents a weak entity type. It depends on the entity type *Employee* through the identifying relationship type *depdents_of*.

3.1.2 Relationships, relationship sets, relationship types, and roles

A relationship links entities from the participating entity types. Relationship sets are identified by relationship types that describe the semantic of the relationships in them. Examples for relationship types are *manages* or *works_for*. It might happen that a relationship has multiple participating entities from the same entity type. In this case, the entity types are equipped with role names to resolve the ambiguity. A

supervisor relationship is an example for a relationship between two entities from the same entity type *Employee*. To uniquely identify the participating employees, we use the two roles *Supervisor* and *Supervisee*. In a graphical representation of an ER schema, relationship types are shown as diamonds and there is an edge between the relationship type and each participating entity type. In Fig. 6, the relationship types *manages*, $works\_for$, $works\_on$, and *controls* can be found.

3.1.3 Attributes, values and value sets

So far, we know how to model relationships and entities. However, the question of how we store information for an entity or relationship is still open. How do we store the information that a car is a red VW? Chen therefore proposed attributes which are mathematical functions that map an entity or relationship to a value from a value set or a tuple with values from different value sets (composite attribute). A value set is comparable to a domain of an attribute in the relational model and includes all valid values. Examples for attributes are *name*, *address*, and *color*, and examples for value sets are *integer* and *numeric*. Entities as well as relationships can have an arbitrary number of attributes. In a graphical representation of an ER schema, an attribute is represented by an oval and is connected to its associated entity type, respectively relationship type. There are two types of attributes that are special: The first one are composite attributes, which can themselves consist of further subattributes. An example for a composite attribute in Fig. 6 is an employee's *name* which consists of the subattributes $fname$, *minit*, and *lname*. The other type of special attributes are multivalued attributes. They are represented by double-lined ovals. Multivalued attributes allow multiple values from the attribute's value set to be present for a single entity or relationship. We call the value set of a multivalued attribute the domain of the multivalued attribute. Fig. 6 shows, for example, the multivalued *locations* attribute of a department. For our EER model, we define the following standard

value sets which should be familiar from the relational model and SQL:

- Small integer, integer, big integer
- Numeric - with parameters *digits* and *decimals*
- Char - with parameter *maximum length*
- Varchar - with parameter *maximum length*
- Date
- Timestamp

3.1.4 Entity keys and duplicate treatment

Chen also introduced an entity key or entity identifier concept which is basically identical to the key concept of the relational model. An entity key is an attribute or a group of attributes that uniquely identify all entities in an entity set. Among all possible entity keys, the database designer has to choose one primary key. The primary key has to be minimal which means that no subset of the primary key attributes is allowed to be a key. In the graphical representation of an ER schema, primary key attributes are underlined. Each entity type must have an identifier. While there is no key for a relationship directly, each relationship can be uniquely identified by the primary keys of the participating entities. As a consequence, our data model is set-oriented.

3.1.5 Relationship constraints and value set constraints

Chen further proposed different types of constraints for his ER model. The most important ones are the relationship constraints. He originally proposed the *cardinality ratio* notation, in which one defines if a given relationship is a *1-to-1*, *1-to-n* or *n-to-m* mapping. We decided to use the more accurate *min-max* notation, in which one can specify for each relationship type the minimum number of relationship instances an

entity from a participating entity type must participate in and the maximum number it can participate in. Beside relationship constraints, Chen proposed data integrity constraints for value sets. We decided to implement three types of these constraints to allow the definition of value sets:

1. *Representative constraint*: User defined value sets based on primitive value sets. For example, *salary* may be defined as a value set over the basic type *integer*. This is similar to the domain constraint in SQL.

2. *Format constraint*: A regular expression can be defined that must be matched by a value for a self-defined value set. $m|f$ can be used for example to restrict a self-defined *gender* value set which is derived from *char(1)*.

3. *Range constraint*: A range can be defined to restrict the values in a self-defined value set. The self-defined *salary* value set can be range restricted to [10000, 100000] to guarantee that each employee earns at least 10,000 but not more than 100,000 per year.

3.2 Extensions in our targeted EER model

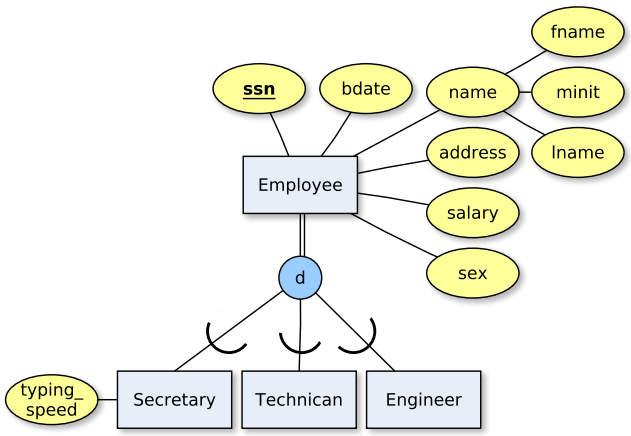


Figure 7: Exemplary *Employee* specialization

3.2.1 Generalization and specialization of entity sets

To understand the concept of generalization and specialization, we introduce the concept of subclasses, superclasses, and inheritance first. Sometimes, it is necessary to subgroup an entity type. The *Employee* entity set could be further split into *Secretary*, *Technican*, and *Engineer* to model different types of employees. We call *Employee* the superclass and *Secretary*, *Technican*, and *Engineer* subclasses. It is important to understand, that an entity of a subclass entity set is automatically member of the superclass entity set. The reverse does not hold. A concept closely related to superclasses and its corresponding subclasses is inheritance. A subclass entity type automatically inherits all attributes of the superclass entity type. Furthermore, the subclass entity inherits all relationships in which the entity as a superclass member participates. Beside the attributes and relationships a subclass entity inherits from its superclass, it can have additional attributes and relationships which only the given subclass entity possesses. Specialization is a top-down approach to model superclass-subclass relations. It means defining a set of subclasses for an existing entity type. In the above example, *Secretary*, *Technican*, and *Engineer* is one specialization of the superclass *Employee* entity type. One superclass entity type can have multiple specializations. It is also possible that two specializations share a common subclass entity type. Fig. 7 shows the graphical representation of an *Employee* specialization. Generalization is the reverse process to specialization and describes a bottom-up approach. For several subclass entity types a superclass entity type is defined. We only use the concept of specialization in the ERSQL *data definition language (DDL)*. A participation constraint can be defined for a specialization. Sometimes, each superclass entity must be a member of some subclass in a specialization. In this case, we talk about a total specialization. Otherwise, we call the specialization partial. A double line in the graphical representation of a specialization indicates a

total specialization (e.g. the employee specialization shown in Fig. 7). Another important constraint is whether a specialization is disjoint or overlapping. For disjoint specializations, a superclass entity is only allowed to be member of one subclass of the specialization. For an overlapping specialization, it might be member of multiple subclasses. ERSQL supports these specialization constraints in its DDL. Graphically, a disjoint specialization is represented by a *"d"* in the specialization node. For an overlapping specialization, we use *"o"*.

3.2.2 Null values, not-null- and default-value-constraints

During the development of our prototype system, we realized that our EER model lacks a clear definition of what the current value of an attribute is if no value currently exists for that attribute for a given entity or relationship. We decided to introduce *null* for this case. If no value is provided for an attribute, its value becomes *null*. We further extended the constraints a database designer can impose on an attribute because there might be situations, in which a value for an attribute should be provided or at least a default value should be used if no value is provided. Therefore, we introduced a *not-null-* and a *default-value-constraint*. These constraints are well known from the SQL DDL.

3.3 Aggregation with multivalued attributes

Gogolla and Hohenstein [11] claimed in their EER calculus proposal that aggregation and grouping should be supported by an EER calculus. We take a different stand because it is easier to incorporate aggregation into our EER model directly. Multivalued attributes fit quite well with the concept of aggregation as they can contain multiple values for a single attribute. The reader might wonder why this should be helpful towards aggregation and grouping of data. This will become obvious when the relationship join is presented in the next section. We allow aggregation functions such as *count*, *average*, *min*, or *max* for all multivalued attributes. These functions are also

Table 1: Aggregation function - value-set matrix

	avg	**cnt**	**min**	**max**	**sum**	**hmean**	**median**	**mode**
Integer	■	■	■	■	■	■	■	■
Numeric	■	■	■	■	■	■	■	■
Char		■	■	■				■
Varchar		■	■	■				■
Date		■	■	■				■
Timestamp		■	■	■				■
Composite		■						
Multivalue		■						

defined for all components of a multivalued composite attribute. Let the values for a given employees salary history be $\{10000, 25000, 30000, 27500, 50000\}$. Our EER model supports for example the following operations on the *salary_history* attribute without the need for any algebra operator:

- *salary_history.cnt*() evaluates to 5
- *salary_history.avg*() evaluates to 28,500
- *salary_history.min*() evaluates to 10,000
- *salary_history.max*() evaluates to 50,000

Our data model further supports *sum*, *hmean* (harmonic mean), *median*, and *mode*. Table 1 provides an overview of the aggregation functions that are defined for our standard value sets. A black square indicates that an aggregation function is defined for a value set.

3.4 Notation

We will use the following notation throughout the remaining work:

- We use lowercase variables to refer to a single entity, e.g. $e \in E$ with E being an entity type.

- We use $r = [(e_1, e_2, ..., e_n), (v_1, v_2, ..., v_m)] \in R$ to refer to a single relationship from the relationship type R. With $e_1, ..., e_n$ being participating entities from the corresponding entity types and $v_1, ..., v_m$ being the values for R's attributes.
- We use $a(e)$ to return the current value(s) for an attribute a of the entity e.
- We use $a(r)$ to return the current value(s) for an attribute a of the relationship r.
- We use $m(v)$ to return the current value of a component m of a multivalued composite attribute. v is one of the values of the multivalued attribute for a given entity or relationship.
- The notation for multivalued composite attributes can be applied recursively in the case of nested multivalued attributes.

Fig. 8 provides an exemplary EER schema. We use it for the examples in the subsequent work. It includes all concepts of our targeted EER model. *SalariedManager* is an example for a shared subclass. The min-max notation is given on each edge that connects a relationship type and a participating entity type. The following example should clarify how they have to be read: Consider the relationship type *works_on*. The min-max notation defines that an employee must works on at least 1 project but can work on at most 10 projects. A project must have at least 3 employees working on it but can have arbitrarily many working on it. Furtermore, the schema shows a total, disjoint specialization of *Employee* into *HourlyEmployee* and *SalariedEmployee*. That means each employee is either an hourly employee or a salaried employee.

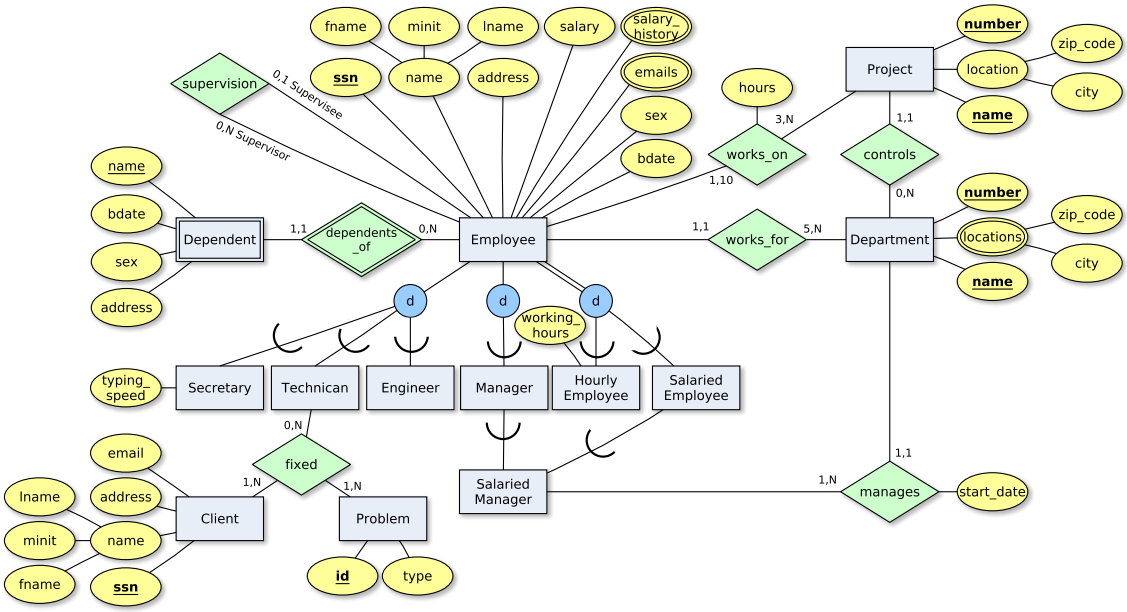


Figure 8: Exemplary EER schema

CHAPTER IV

THE EER ALGEBRA

The functional primitives for our ERDBMS are a slightly adapted version of the algebra proposed by Parent and Spaccapietra [23, 24]. We chose their work because it is a perfect fit with the characteristics of our EER model. It combines the concept of joins with multivalued attributes and avoids unnecessary flattening of an algebra result. Moreover, their algebra is general enough to work with our targeted EER model. We agree with their opinion that an algebra must only be defined for entity types, which means, that the operands of an operator are entity types only, and, that the result of an operator is a new entity type. Thus, the algebra is closed. Relationships are used in the algebra to join two operand entity types in a relationship join. We have slightly adapted the proposed algebra because our EER model is set oriented and, therefore, we do not need a compression operator for duplicate elimination. Moreover, we equipped the projection operator with some additional functionalities and extended the relationship join by quantifiers. We also saw the need for four additional operators to have a sound basis for our high-level query language ERSQL. They are a casting operator, a collapse operator, an additional cartesian product operator, and a merge-join operator. We will discuss all 13 operators in detail now.

4.1 Nested cartesian product (binary) $E_1 \vec{\times} E_2$

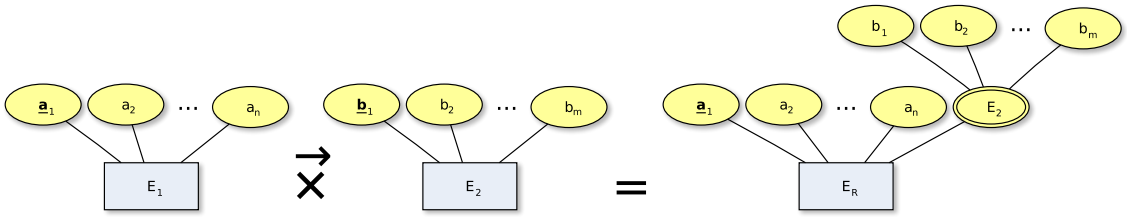


Figure 9: Nested cartesian product - ER transform schema

This operator is used to combine two entity types which are not related by any

relationship type or which should not be related by an existing relationship type between them. Fig. 9 illustrate the schema of the resulting entity type. The entity type E_1 is extended by a multivalued composite attribute E_2. The components of E_2 are the attributes of the operand entity type E_2. The population of the multivalued E_2 attribute for $e \in E_1$ consists of all entities in the operand entity type E_2. Basically, every entity $e \in E_1$ *gets the whole population of* E_2 *nested in a new multivalued attribute* E_2. The nested cartesian product is neither transitive nor associative. (*Name in the original paper: Cartesian Product)*

4.2 Cartesian product (binary) $E_1 \times E_2$

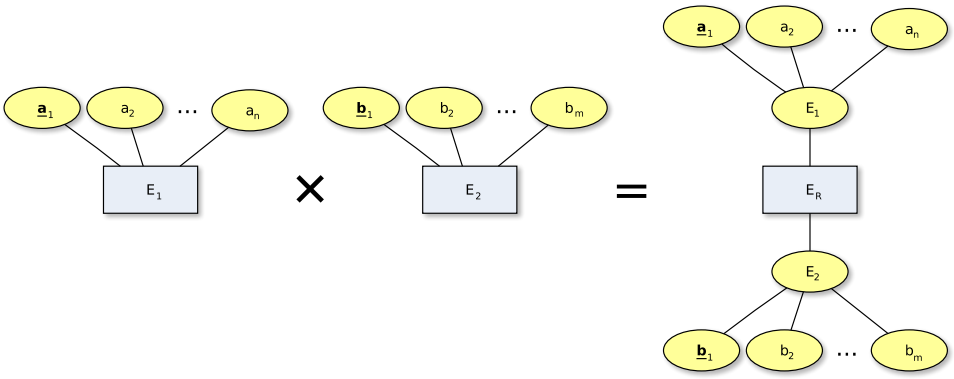


Figure 10: Cartesian product - ER transform schema

The cartesian product is also used to combine two entity sets which are not related by any relationship type or which should not be related by an existing relationship type between them. It is basically the same as its counterpart in the relational algebra. The attributes of E_1 are combined in a composite attribute E_1. The same happens to the attributes of E_2. We call this technique *entity packing* and it is used to avoid attribute naming ambiguities. If an entity set is already packed, it is not packed again. Fig. 10 illustrates our packing technique. We decided to include the cartesian product to have a sound basis for our ERSQL query language because it is transitive and associative:

1. Transitivity: $A \times B = B \times A$

2. Associativity: $(A \times B) \times C = A \times (B \times C)$

4.3 Relationship join (n-ary) $E_1 \bowtie_R (E_2, ..., E_n)$

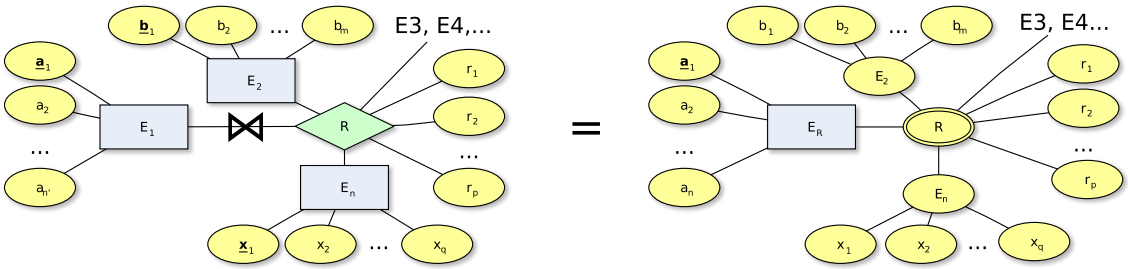


Figure 11: Relationship join - ER transform schema

The relationship type R defined among n entity types is used to join the entity type E_1 with the entity types $E_2, ..., E_n$: $E_1 \bowtie_R (E_2, ..., E_n)$. Fig. 11 illustrates this scenario. The result of the join is to extend the entity set E_1 *by a multivalued composite attribute* R. The components of R are the attributes of the relationship type R and the composite attributes $E_2, ..., E_n$. The components of E_i are the attributes of the operand entitiy type E_i, $\forall i \in [2, ..., n]$. Using only a subset of E_2, ..., E_n on the right side of the join is also possible. Whether an entity from the left operand entity set qualifies for the resulting entity set E_R depends on the mode it is executed in and the quantifiers that are used for the right side entity types. We differentiate between the *standard* join mode and the *outer* join mode. We will provide a formal definition of them shortly. Each operand on the right side can be qualified with an $\exists$ or $\forall$ quantifier. If no quantifier is provided, an $\exists$ quantifier is implicitly assumed. A single relationship join is fixed in its mode but can have different quantifiers on the right side. The population of the multivalued attribute R for $e \in E_1$ can be computed by following all relationships in R, e paritcipates in, and storing the corresponding values of the relationship attributes and all participating entities for the relationship instance. The

set of all such composite attributes builds the population of the multivalued composite attribute R for e. The result entity inherits all relationships e participates in. This is important for further relationship joins. An entity e, extended by R, qualifies for the result entity set if the join was *successful*. Whether a join is considered successful depends on the used quantifiers on the right side of the join. We will introduce a formal definition of a *successful join* shortly when we introduce the quantifiers in detail. It is important to understand that a join for an entity $e \in E_1$ might not be successful even if join partners can be found.

4.3.1 Quantifiers for relationship join

Parent et al. [23, 24] implicitly used an $\exists$ quantifier for all entity types on the right side of a relationship join. That means a join of an entity $e \in E_1$ with $E_2, ..., E_n$ is considered successful, if there is a join partner for each of the right side entity types or in other words, if there is a relationship in R that connects $e \in E_1$ with participating entities from all of the right side entity types. We extended the relationship join by a $\forall$ quantifier. If a right side entity type E_i with $i \in [2, ..., n]$ is qualified with $\forall$, the relationship join of $e \in E_1$ with E_i is considered successful, if e is joinable with all $e' \in E_i$. The right side entity type of a relationship join can be qualified with a mix of $\exists$ and $\forall$ quantifiers. A join of $e \in E_1$ with such mixed qualified entity types $E_2, .., E_n$ is considered successful, if the join of e with each entity type E_i, $\forall i \in [2, ..., n]$, is successful with the given quantifiers. The following examples should clarify any confusion: $Technican \bowtie_{fixed} (\forall Client, \exists Problem)$ returns all technicans (with their clients and fixed problems) who have fixed at least one problem and who have fixed some problem for all clients. $Technican \bowtie_{fixed} (\forall Client, \forall Problem)$ returns all technicans, who have fixed all instances of problems and who have fixed every problem instance for all clients.

4.3.2 Join mode

The examples we used to illustrate the quantifier definitions assumed that the entities from the left operand entity set of the join are omitted if they are not successfully joinable. We call this join mode the *standard* join mode. The *outer* join mode on the other hand keeps entities of the left entity set in the resulting entity set even if they are not successfully joinable. For such entities, the multivalued composite attribute, that is generated by the relationship join, is empty. We use the $\ltimes$ to illustrate an outer join mode: $Employee \ltimes_{works\_on} Project$ returns all employees with the projects they work on. Employees, that do not work on any project, are kept in the resulting entity set.

4.4 Merge join (binary) $E_1 \bowtie^{\succ}_{x} E_2$

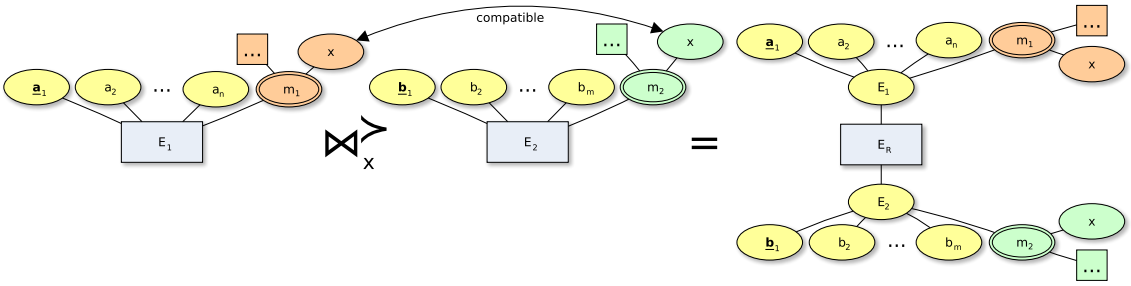
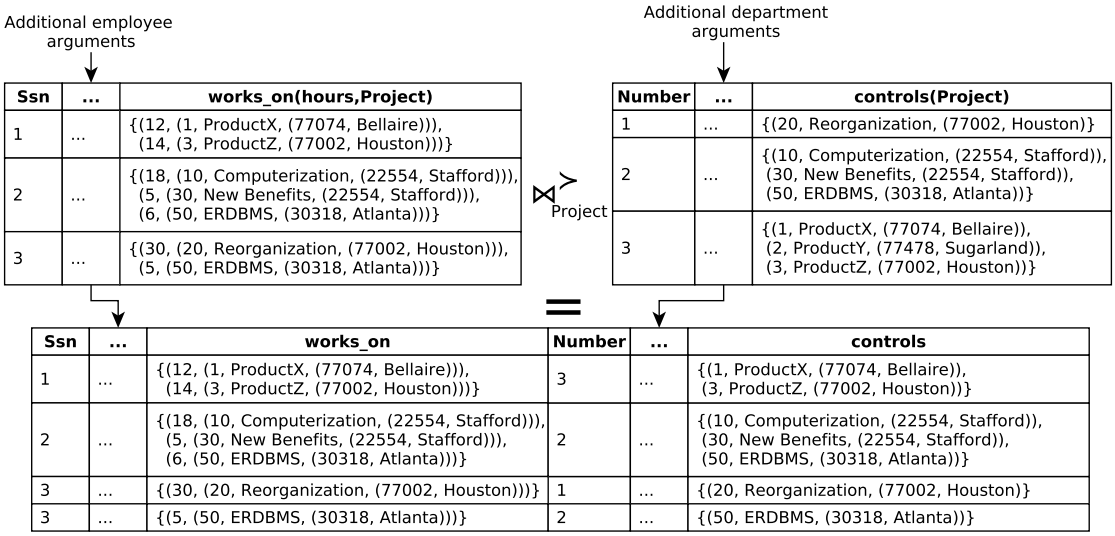


Figure 12: Merge join - ER transform schema

The merge join is used to join two operand entity types based on a compatible attribute that is part of a multivalued attribute in both entity types. Compatible means the same name and value-set (also for components in case of a composite attribute x). Usually, such compatible attributes are the results of joining the same entity type on the right side in a prior relationship join. The schema of the resulting entity type of a merge join can be seen in Fig. 12. *Entity packing* is used again to avoid naming ambiguities. To calculate the population of the result entity type after a merge join, we first introduce the concept of a *multivalued attribute join*.



Ssn	...	works_on(hours,Project)
1	...	{(12, (1, ProductX, (77074, Bellaire))), (14, (3, ProductZ, (77002, Houston)))}
2	...	{(18, (10, Computerization, (22554, Stafford))), (5, (30, New Benefits, (22554, Stafford))), (6, (50, ERDBMS, (30318, Atlanta)))}
3	...	{(30, (20, Reorganization, (77002, Houston))), (5, (50, ERDBMS, (30318, Atlanta)))}

Number	...	controls(Project)
1	...	{(20, Reorganization, (77002, Houston)}
2	...	{(10, Computerization, (22554, Stafford)), (30, New Benefits, (22554, Stafford)), (50, ERDBMS, (30318, Atlanta))}
3	...	{(1, ProductX, (77074, Bellaire)), (2, ProductY, (77478, Sugarland)), (3, ProductZ, (77002, Houston))}

Ssn	...	works_on	Number	...	controls
1	...	{(12, (1, ProductX, (77074, Bellaire))), (14, (3, ProductZ, (77002, Houston)))}	3	...	{(1, ProductX, (77074, Bellaire)), (3, ProductZ, (77002, Houston))}
2	...	{(18, (10, Computerization, (22554, Stafford))), (5, (30, New Benefits, (22554, Stafford))), (6, (50, ERDBMS, (30318, Atlanta)))}	2	...	{(10, Computerization, (22554, Stafford)), (30, New Benefits, (22554, Stafford)), (50, ERDBMS, (30318, Atlanta))}
3	...	{(30, (20, Reorganization, (77002, Houston)))}	1	...	{(20, Reorganization, (77002, Houston)}
3	...	{(5, (50, ERDBMS, (30318, Atlanta)))}	2	...	{(50, ERDBMS, (30318, Atlanta))}

Figure 13: Example of a merge join

Definition (Multivalued attribute join based on matching x from participating multivalued attributes)**.** Let e_1 be an entity from E_1 and e_2 be an entity from E_2. The population of the multivalued attribute join based on the compatible attribute x from multivalued attributes $m_1(e_1)$ and $m_2(e_2)$ is defined as:

$$m_1(e_1) \bowtie_x m_2(e_2) = \{v \mid v \in m_1(e_1), s.t.\ \exists v' \in m_2(e_2),\ x(v) = x(v')\}$$

Entities $e_1 \in E_1$ and $e_2 \in E_2$ are joinable if the multivalued attribute join $m_1(e_1) \bowtie_x m_2(e_2)$ is not empty. The resulting entity is generated by keeping the values of e_1 for all attributes of E_1 with m_1 reduced to $m_1(e_1) \bowtie_x m_2(e_2)$ and by keeping the values of e_2 for all attributes of E_2 with m_2 reduced to $m_2(e_2) \bowtie_x m_1(e_1)$. For *ERSQL*, we have extended our merge join operator to join based on multiple compatible attributes $x_1, ..., x_n$ which are all part of distinct multivalued attributes. A join of e_1 and e_2 is successful, if all multivalued attribute joins based on all x_i, $\forall i \in [1, ..., n]$, are not empty. The operator is mainly used to answer queries like: Give me all pairs of employees and departments such that the department controls at least one project the employee works on: $(Employee \bowtie_{works\_on} Project) \bowtie^{\succ}_{Project}$

($Department \bowtie_{controls} Project$). Fig 13 illustrates this specific merge join. The top 2 tables in Fig. 13 show the left and right argument relations of the merge join. The examples at the end of the section show some more usecases.

4.5 Collapse (unary) $\xi_c(E)$

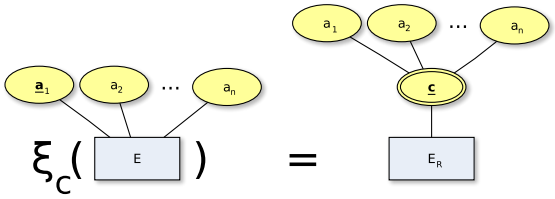


Figure 14: Collapse operator - ER transform schema

The collapse operator collapses the entity set of an entity type E into a single entity by introducing a new multivalued composite attribute c which has all attributes of E as its components. An entity in E becomes a value for c. $\xi_e(\sigma_{sex='f'}(Employee))$, for example, returns a single entity with a multivalued composite attribute e which contains all female employees. The entity type name of the source entity type is preserved. Fig. 14 shows the transformation schema of the collapse operator.

4.6 Selection (unary) $\sigma_p(E)$

The selection operator filters entities from the operand entity set based on predicates expressed over the attributes of that entity type. The operator is quite similar to its relational counterpart. The main difference between them results from the fact, that our selection operator has to deal with multivalued and composite attributes as well. We will first introduce the predicates that can be expressed on monovalued attributes, before we present the predicate formulation techniques for multivalued attributes. Two predicates can be combined into a single predicate by the binary $\wedge$ and $\vee$ operators. Parentheses can enforce an evaluation order among the linked predicates.

4.6.1 Monovalued selection predicates

Predicates for monovalued attributes can be expressed using the arithmetic operators $=$, $<>$, $>$, $\geq$, $<$, and $\leq$. These comparison operators can be used to relate two monovalued attributes which are value-set-compatible or to compare a monovalued attribute with a constant. The constant must be a value from the attribute's value set, that it is compared to. An entity qualifies for the resulting entity set if the selection predicate p evaluates to true for its current attribute values. The expression $\sigma_{location.city='Atlanta' \vee location.city='Munich'}(Project)$ filters projects, that are located in Atlanta or Munich. The aggregate functions we defined in Section 3.3 for all multivalued attributes generate a monovalued attribute and, thus, can also participate in the selection predicate as a standard monovalued attribute. The following expression, for example, returns all departments with at least 3 locations: $\sigma_{locations.cnt()=3}(Department)$, while $\sigma_{salary\_history.avg()>30{,}000}(Employee)$ returns all employees who have earned on average more than 30,000 during their working life.

4.6.2 Multivalued selection predicates

To express a predicate on a multivalued attribute, we follow the idea of Parent and Spaccapietra [24] to use $\exists$ and $\forall$ quantifiers. Such a multivalued selection predicate has either the form $\exists_i x \in mv\_attr\ (p_i)$ or $\forall x \in mv\_attr\ (p_i)$ where mv_attr is a path to a multivalued attribute and p_i is a selection predicate itself. The bind variable name is arbitrary and does not have to be x. The bind variable is used to iterate over all values of the multivalued attribute during evaluation. The inner predicate p_i can be expressed on all attributes of the source entity type and on the bind attribute x which contains a single value of the multivalued attribute at a time. The multivalued predicates can be applied recursively if x consists of another multivalued attribute. $\exists_i x \in mv\_attr\ (p_i)$ evaluates to true if there are i values in mv_attr for a given entity e that make p_i true. $\forall x \in mv\_attr\ (p_i)$ is true if all values for e make

p_i true. $\sigma_{\forall x \in salary\_history(x>=50000 \wedge x<=100000}(Employee)$ returns employees that have a multivalued set for salary_history all of whose elements are between 50,000 and 100,000. It is also allowed to use an attribute path to a component of a multivalued composite attribute. The bind variable binds itself to the value of the component: $\sigma_{\exists_1 x \in locations.city(x='Atlanta' \vee x='Houston')}(Department)$ returns all departments that have a locations multivalued attribute that contains Atlanta or Houston (or both).

4.7 Reduction (unary) $\chi_p(E)$

While the selection operator only filters entities $e \in E$ based on the predicate p, a reduction can be used to remove values from a multivalued attribute that fulfill the predicate p. The predicate p can be written in the following form to reduce a multivalued attribute: $x \in mv\_attr(p_i)$. The inner predicate p_i can be any arbitrary predicate, similar to the inner predicates in the multivalued selection predicates. It can be expressed on all attributes of the source entity set and on the bind variable x. $\chi_{x \in salary\_history(x>100000 \vee x<10000)}(Employee)$ removes all salaries that are bigger than $100,000$ or smaller than $10,000$ from the salary history of an employee. If a reduction is expressed on an attribute path with multiple levels of multivalued attributes, the reduction is performed for every single instance of the multivalued attribute that should be reduced: $\chi_{x \in salary\_history(x>100000 \vee x<10000)}(Department \bowtie_{works\_for} Employee)$ reduces the salary history of every single employee of a department.

4.8 Projection (unary) $\Pi_A(E)$

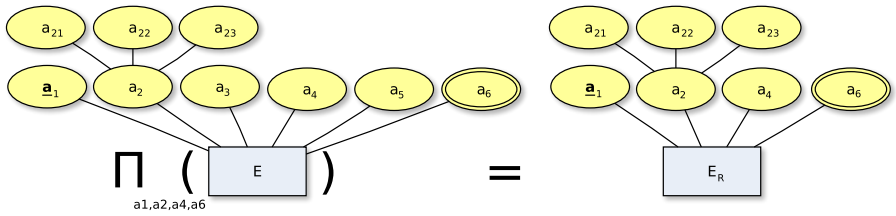


Figure 15: Projection operator - ER transform schema

This operator is used to project an entity type onto a subset A of its attributes. It is basically the equivalent to the projection operator in the relational algebra. Fig. 15 illustrates the projection of an entity type E_1 onto the attributes a_1, a_2, a_4, and the multivalued attribute a_6. The illustrated projection can be written as $\Pi_{a_1,a_2,a_4,a_6}(E)$. If the key attributes are partially or completely omitted, the projection operator eliminates possible duplicates. It is important to understand, that omitting any key attribute from an entity type E destroys all connections to relationships types E participates in and further relationship joins are no longer possible. Our projection operator is more powerful than its relational counterpart as it can be used to un-nest components in composite attributes (also for multivalued composite attributes). The following examples illustrate the un-nesting functionalities: $\Pi_{name(fname,lname)}(Employee)$ projects the composite *name* attribute onto its first and last component. $\Pi_{name.fname}(Employee)$ gets the firstname component of *name*. Furthermore, the projection operator can be used to project a multivalued attribute onto the monovalued attribute that is generated by an aggregation function. The following example returns, for example, the average salary every employee has earned in its working life: $\Pi_{salary\_history.avg()}(Employee)$.

4.9 Casting (unary) $\Phi_C(E)$

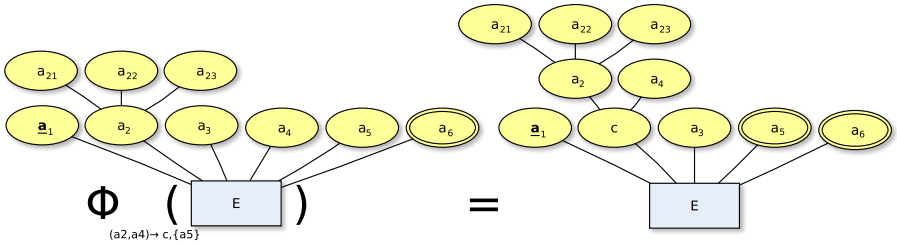


Figure 16: Casting operator - ER transform schema

The casting operator is used to compose multiple attributes into a composite attribute or to transform a monovalued attribute into a multivalued one. It builds

the counterpart to the un-nesting functionalities of the projection and can be used to make two entity types compatible for set operations like union, intersection, or difference. Fig. 16 illustrates the casting of the attributes a_2 and a_4 of E into the composite attribute c. Moreover, it shows the transformation of the monovalued a_5 attribute into a multivalued attribute with at most a single value for each entity.

4.10 Renaming (unary) $\rho_R(E)$

The rename operator is equivalent to its relational counterpart and can be used to rename attribute names and entity type names to resolve naming ambiguities. It is also possible to rename a (multivalued) component of a composite attribute: $\rho_{name.fname \rightarrow firstname}(Employee)$.

4.11 Union (binary) $E_1 \cup E_2$

Union is the first of three set operations, we define for our algebra. It requires two compatible entity types as operands. Compatible means that they have the same amount of attributes and that there are distinct pairs of value-set compatible attributes with one attribute from E_1 and one from E_2. The union operator calculates the set union of E_1 and E_2. Possible duplicates are eliminated. The naming (attributes and entity type) of the left operand entity type is preserved.

4.12 Intersection (binary) $E_1 \cap E_2$

The intersection operator is the second of our three set operators. It also requires two compatible entity types as operands. It is used to calculate the set intersection of E_1 and E_2. The naming (attributes and entity type) of the left operand entity type is preserved.

4.13 Difference (binary) $E_1 - E_2$

The last of our three set operators is the difference operator. It requires, like all our set operators, compatible operand entity sets. It calculates the set difference of E_1 and E_2. The naming (attributes and entity type) of the left operand entity type is preserved.

4.14 Examples of EER algebra queries

The following queries should give a good idea about how the presented operators can be used to answer queries (based on schema in Fig. 8):

Q1: List the distinct first names of all employees:

$\Pi_{name.fname}(Employee)$

Q2: List all employees with the projects they work on:

$Employee \bowtie_{works\_on} Project$

Q3: Get the number of managers, their average salary, and their last names:

$\Pi_{M.cnt(),M.salary.avg(),M.name.lname}(\xi_M(Manager))$

Q4: Get the ssn of all engineers that work for the department that controls the *ERDBMS* project:

$\Pi_{ssn}(Engineer \bowtie_{works\_for} (Department \bowtie_{controls} (\sigma_{name='ERDBMS'}(Project))))$

Q5: Get engineers, who work on all projects the *Research* department controls:

$Engineer \bowtie_{works\_for} \forall(Project \bowtie_{controls} (\sigma_{name='Research'}(Department)))$

Note that the $\forall$ quantifier is applied here to the list of projects that are returned from the join that are controlled by the research department.

Q6: Get the ssn of all employees that are neither technicans nor hourly employees:

$\Pi_{ssn}(Employee) - (\Pi_{ssn}(Technicans) \cup \Pi_{ssn}(HourlyEmployee))$

Q7: Get the name, number, and locations of all department and projects:

$Department \cup \Phi_{\{location\}}(Project)$

Note the use of the cast operator that converts location from monovalued to a multivalued attribute and makes Department *and* Project *union compatible.*

Q8: List all secretaries with the department she/he works for, the project she/he works on, and her/his supervisor (if they have one):

$((Secretary \bowtie_{works\_for} Department) \bowtie_{works\_on} Project)_{Supervisee} \rtimes_{supervision} Employee_{Supervisor}$

Q9: Get all department names, the maximum salary among its technicans, and the technican names:

$\Pi_{name,works\_for.Technican.salary.max(),works\_for.Technican.name}(Department \rtimes_{works\_for} Technican)$

Note that the Technician.name is a multivalued attribute generated from the join that contains names of all technicians.

Q10: Get the name of departments that only control projects which are located in the same cities the department has locations in:

$\Pi_{name}(\sigma_{\forall x \in controls.Project.location\ (\exists_1 y \in locations\ (y.city=x.city))}(Department \bowtie_{controls} Project))$

Q11: Get all pairs of employees such that the employees work for the same department and on at least one Project together:

$\sigma_{E_1.ssn<>E_2.ssn}(((\rho_{Employee \to E_1}(Employee) \bowtie_{works\_for} Department) \bowtie_{works\_on} Project)$

$\bowtie^{\succ}_{Department,Project} ((\rho_{Employee \to E_2}(Employee) \bowtie_{works\_for} Department) \bowtie_{works\_on} Project))$

Note here how we conduct a merge join among two Employee *entity-sets on the nested multivalued attributes* Department *and* Project.

Q12: Get all employee-department pairs such that the employee works on at least one project for at least 10 hours, that the department controls:

$\chi_{x \in works\_on.hours(x<10)}(Employee \bowtie_{works\_on} Project) \bowtie^{\succ}_{Project} (Department \bowtie_{controls} Project)$

Note how we used the reduction operator by eliminating projects on which an employee works for less than 10 hours.

CHAPTER V

THE ERSQL QUERY LANGUAGE

In this section, we introduce our high-level declarative query language *ERSQL*. We decided to design ERSQL based on SQL because the success of the relational model is closely related to the simple and declarative characteristics of SQL. There have been many proposals for high-level querying languages for the ER data model but none of them could become widely accepted and hence none was commercially implemented [9, 16, 17]. They either lack a semantic foundation (algebra or calculus), show navigational characteristics, or they do not incorporate the ER data model in a satisfactory manner. For ERSQL, we tried to export the semantic richness of the ER data model directly to the user. As an example, joins are no longer expressed by join predicates based on attribute names, but by the name of relationship types which link entity types. Our major design goal was: make ERSQL as powerful as possible while keeping it as simple as possible. Similar approaches for declarative query languages for the ER model have been proposed by Wuu [28] and Hohenstein et al. [14]. But they have a rather complex syntax, and in some sense their semantic foundations do not incorporate the ER model in a satisfactory manner. Wuu's SERQL language allows subqueries similar to SQL. During the development of our ERSQL language, we have realized that there is no need for subqueries if you exploit the non-1-NF characteristic of the ER model. We will provide more details on this at the end of the chapter. Moreover, almost all proposals for a high-level ER query language introduce a query language for data retrieval only. To our knowledge, no proposal covers a *data definition language (DDL)* and a *data manipulation language (DML)*. ERSQL offers both, a comprehensive DDL and a DML which allows inserting, deleting, updating, and retrieving of data.

The remainder of this section is twofold. The first part covers the DDL of ERSQL and the second part provides details on the DML. The second part also introduces our canonical translation algorithm that maps a data retrieval query, expressed in ERSQL, into an equivalent EER algebra expression. We present the algorithm in a

step-by-step fashion by showing how the different clauses of an ERSQL query can be mapped into an algebra expression. We use the *Backus-Naur-Form (BNF)* to present the grammer of ERSQL. We omitted the definition of self-explanatory non-terminals like $<name>$ or $<integer>$ to save space. For better understanding, we provide examples for every query type. All examples are based on the EER schema in Fig. 8.

5.1 Data definition language (DDL)

The DDL of ERSQL offers commands to define and remove entity types, relationship types, and value sets. Furthermore, the user can define weak entity types and specializations of entity types.

5.1.1 Create type

⟨*create_type*⟩ ::= '`create type` ' ⟨*name*⟩ ' `from` ' ⟨*representant_type*⟩ '`(`'
[⟨*constraints*⟩]
'`);`'

⟨*representant_type*⟩ ::= '`smallinteger`'
| '`integer`'
| '`biginteger`'
| '`numeric (`' ⟨*digits*⟩ '`,`' ⟨*decimals*⟩ '`)`'
| '`date`'
| '`timestamp`'
| '`varchar (`' ⟨*maxlength*⟩ '`)`'
| '`char (`' ⟨*maxlength*⟩ '`)`'
| ⟨*user_defined_type*⟩

⟨*constraints*⟩ ::= ⟨*format*⟩ '`,`' ⟨*range*⟩
| ⟨*format*⟩
| ⟨*range*⟩

⟨*format*⟩ ::= '`format ''`' ⟨*regular_expression*⟩ '`''`'

⟨*range*⟩ ::= '`range min (`' ⟨*standard_literal*⟩ '`)`' ['` max (`' ⟨*standard_literal*⟩ '`)`']
| '`range`' ['` min (`' ⟨*standard_literal*⟩ '`)`'] '` max (`' ⟨*standard_literal*⟩ '`)`'

⟨*standard_literal*⟩ ::= ⟨*integer*⟩
| ⟨*decimal*⟩
| '`''`' ⟨*string*⟩ '`''`'

The *create type* command creates a user-defined value set. The representative type must either be a standard- or an existing user-defined value set. Within the body of a create type command, the value-set constraints mentioned in section 3.1.5 can be defined. The non terminal *standard_literal* for the min and max values of the range constraint must be a value from the representantive value set. The following Listing shows the definition of the *salaryType* and *emailType* value sets of our examplary EER schema:

Listing 5.1: Definition of the *salaryType* and *emailType* value sets

```
create type EMailType from Varchar(150) (
    format '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,4}'
);
create type SalaryType from Integer (
    range min(10000) max(100000)
);
```

5.1.2 Create entity

⟨*create_entity*⟩ ::= 'create entity' ⟨*name*⟩ '('
⟨*attribute*⟩ {',' ⟨*attribute*⟩} ','
'key (' ⟨*attribute_name*⟩ {',' ⟨*attribute_name*⟩} ')'
');'

⟨*attribute*⟩ ::= ⟨*attribute_name*⟩ ' ' ⟨*attribute_type*⟩ [' not null'] [' default' ⟨*literal*⟩]

⟨*attribute_type*⟩ ::= ⟨*representant_type*⟩
| 'multivalued (' ⟨*attribute_type*⟩ ')'
| 'composite (' ⟨*attribute*⟩ ',' ⟨*attribute*⟩ {',' ⟨*attribute*⟩} ')'

⟨*literal*⟩ ::= ⟨*standard_literal*⟩
| ⟨*composite_literal*⟩
| ⟨*multivalue_literal*⟩

⟨*composite_literal*⟩ ::= '(' ⟨*literal*⟩ ',' ⟨*literal*⟩ {',' ⟨*literal*⟩} ')'

⟨*multivalue_literal*⟩ ::= '{' ⟨*literal*⟩ {',' ⟨*literal*⟩} '}'

An entity type must have at least one attribute and a key. An attribute can be not null constrained and a default value can be provided. Key attributes are automatically not null constraint. The non-terminal *literal* stands for the default value.

Constants for composites begin with *"("* and end with *")"*. The component constants are defined in between in a comma-separated list: *(*$component_1$*,* $component_2$*,* $component_3$*)*. Constants for multivalued attributes begin with *"{"* and end with *"}"*. The value constants are listed in between seperated by a comma: {$value_1$, $value_2$, $value_3$}. The following Listing shows the definition of the *Employee* entity type of our examplary EER schema:

Listing 5.2: Definition of the *Employee* entity type

```
create entity Employee (
    ssn Integer,
    bdate Date,
    emails Multivalued ( EMailType ) not null,
    name Composite (
        fname Varchar(50) not null,
        minit Char(1),
        lname Varchar(50) not null
    ),
    address Varchar(150) not null,
    salary SalaryType not null default 30000,
    salary_history Multivalued ( SalaryType ),
    sex GenderType not null,
    key (ssn)
);
```

5.1.3 Create relationship

⟨*create_relationship*⟩ ::= ‘`create relationship` ’ ⟨*name*⟩ ‘`(`’
⟨*relationship_entities*⟩
[‘`,`’ ⟨*relationship_attributes*⟩]
‘`);`’

⟨*relationship_entities*⟩ ::= ‘`entities (`’ ⟨*entity*⟩ {‘`,`’ ⟨*entity*⟩} ‘`)`’

⟨*relationship_attributes*⟩ ::= ‘`attributes (`’ ⟨*attribute*⟩ {‘`,`’ ⟨*attribute*⟩} ‘`)`’

⟨*entity*⟩ ::= ⟨*entity_type*⟩ [‘ ’ ⟨*role_name*⟩] [‘ `min (`’ ⟨*integer*⟩ ‘`)`’] [‘ `max (`’ ⟨*integer*⟩ ‘`)`’]

A relationship type must have at least two participating entity types. The min-max constraints and, if necessary, a unique role name can be defined with each entity

type. Furthermore, the *create relationship* command allows the definition of additional relationship attributes. The following Listing shows the definition of the *supervision* and *works_on* relationship type of our examplary EER schema:

Listing 5.3: Definition of the *supervision* and *works_on* relationship type

```
create relationship supervision (
    entities (
        Employee Supervisee max(1),
        Employee Supervisor
    )
);
create relationship works_on (
    entities (
        Employee min(1) max(10),
        Project min(3)
    ),
    attributes (
        hours Integer not null default 5
    )
);
```

5.1.4 Create weak entity

⟨*create_weak_entity*⟩ ::= 'create weak entity ' ⟨*name*⟩ '('
'defined by (' ⟨*identifying_relationship*⟩ {',' ⟨*identifying_relationship*⟩} ')'
',' ⟨*attribute*⟩ {',' ⟨*attribute*⟩} ','
'partial key (' ⟨*attribute_name*⟩ {',' ⟨*attribute_name*⟩} ')'
');'

⟨*identifying_relationship*⟩ ::= ⟨*create_relationship*⟩ ' as ' ⟨*role_name*⟩
| ⟨*create_relationship*⟩

The existence of a weak entity is coupled with the existence of strong entities. The relationship between a weak and a strong entity is called an identifying relationship. Therefore, at least one relationship type must be defined in the *defined by* clause. Such a relationship type is not allowed to have any own attributes. As a weak entity participates in one relationship per identifying relationship type only, these attributes should belong to the weak entity directly. The weak entity type must not to be listed

in the *entities* clause of such an identifying relationship type. A weak entity set can have multiple identifying relationship types. If the weak entity type should participate with a role name, *"as <role_name >"* can be added immediately after the definition of the identifying relationship type. The key of a weak entity is the combination of the strong entities' keys and its own partial key. The following Listing shows the definition of the weak *Dependent* entity type of our examplary EER schema:

Listing 5.4: Definition of the weak *Dependent* entity type

```
create weak entity Dependent (
    defined by (
        create relationship dependent_of (
            entities ( Employee )
        )
    ),
    name Varchar(50) not null,
    bdate Date,
    sex GenderType,
    address Varchar(150) not null,
    partial key (name)
);
```

5.1.5 Create specialization

⟨*create_specialization*⟩ ::= 'create ' ⟨*participation*⟩ ' ' ⟨*membership*⟩ ' specialization of ' ⟨*entity_type*⟩ '('
⟨*create_entity*⟩ {',' ⟨*create_entity*⟩}
');'

⟨*participation*⟩ ::= 'total'
| 'partial'

⟨*membership*⟩ ::= 'distinct'
| 'overlapping'

The *create specialization* statement creates a set of entity types that are subgroups of the parent entity type. The subclass entity type definitions in the body of the command must not contain its own key definition as the key is already defined by the superclass. Additional attributes can also be defined within the subclass entity type definitions. Shared subclasses can be created by using the same name for a subclass

entity type in multiple specializations which share the same root. The following Listing shows the specialization of the *Employee* entity set of our examplary EER schema into *Secretary*, *Technican*, and *Engineer*:

Listing 5.5: Specialization of *Employee* entity set

```
create partial distinct specialization of Employee (
    create entity Secretary (
        typing_speed Integer not null
    ),
    create entity Technican (),
    create entity Engineer ()
);
```

Appendix A.1 contains the complete schema definition for Fig. 8.

5.1.6 Drop

The drop command removes entity types, relationship types, and value sets from the schema. An entity type cannot be deleted if it participates in any relationship type. If an entity type is dropped, its subclasses are also dropped.

⟨*drop_entity*⟩ ::= '`drop entity` ' ⟨*entity_type*⟩

⟨*drop_relationship*⟩ ::= '`drop relationship` ' ⟨*relationship_type*⟩

⟨*drop_type*⟩ ::= '`drop type` ' ⟨*value_set*⟩

5.2 Data manipulation language (DML)

We tried to design our DML as similar to SQL as possible. Our *insert*, *update*, and *delete* commands are almost identical to their SQL equivalents. We extended them slightly to incorporate multivalued and composite attributes. For the *select* command, we dropped some unnnecessary clauses but also added some new ones.

5.2.1 Insert into

⟨*insert*⟩ ::= '`insert into` ' ⟨*name*⟩ '` set` ' ⟨*attribute_path*⟩ '`=`' ⟨*literal*⟩ {',' ⟨*attribute_path*⟩ '`=`' ⟨*literal*⟩} ';'
| '`insert into` ' ⟨*name*⟩ '`(`' ⟨*attribute_path*⟩ {',' ⟨*attribute_path*⟩} '`) values (`' ⟨*literal*⟩ {',' ⟨*literal*⟩} '`);`'

⟨*attribute_path*⟩ ::= ⟨*string*⟩ {'.' ⟨*string*⟩}

Data can be inserted into entity sets and relationship sets. Inserts can be expressed in two different ways. The examples below illustrate them. It is important that an insert into a relationship set must contain keys for the participating entities. They can be provided using the entity type's role name as the attribute path. If no role name was defined, the entity type must be used. The last two inserts in Listing 5.6 show relationship inserts. An attribute path can be used to insert data for components of a composite attribute (Listing 5.6, query 2). But such an attribute path cannot cross a multivalued attribute. Values for multivalued attributes can only be inserted as a whole.

Listing 5.6: Inserts into entity- and relationship sets

```
insert into Employee set ssn=888665555, bdate='1937-11-10', sex='m', salary=55000,
    emails={'james.borg@borg.com', 'jborg@borg.com'}, name=('James', 'E', 'Borg'),
    address='450 Stone, Houston, TX', salary_history={25000,30000};
insert into Project (name, number, location.zip_code, location.city) values
    ('Reorganization', 20, 77002, 'Houston');
insert into works_on set Employee=888665555, Project=('Reorganization', 20),
    hours=35;
insert into supervision (Supervisee, Supervisor) values (987654321, 888665555);
```

An insert of a weak entity must also contain the primary keys of all strong entities, it depends on. The corresponding identifying relationships are inserted automatically with the weak entity. The attribute name, that can be used to add the key of the strong entity, is the role name of the strong entity type in the corresponding identifying relationship type. If no role name was defined, the entity type must be used.

Listing 5.7: Inserts into weak entity set *Dependent*

```
insert into Dependent set Employee=987654321, name='Jane Doe', sex='f',
    address='291 Berry, Bellaire, TX';
```

If an existing entity should be added to a subclass, the primary key of the existing entity must be specified in the set clause only (see Listing 5.8). If a new entity should

be directly added to a subclass, it can be inserted normally. In both cases, the entity is automatically added to all superclasses.

Listing 5.8: Adding an existing entity to a subclass

```
-- Suppose ssn=987654321 already exists as an employee
insert into HourlyEmployee set ssn=987654321;
```

5.2.2 Update

⟨*update*⟩ ::= ‘`update` ’ ⟨*name*⟩ ‘ `set` ’ ⟨*attribute_update*⟩ {‘`,`’ ⟨*attribute_update*⟩}
[‘ `where` ’ ⟨*qualification*⟩] ‘`;`’

⟨*attribute_update*⟩ ::= ⟨*attribute_path*⟩ ‘`=`’ ⟨*literal*⟩
| ⟨*attribute_path*⟩ ‘`.append(`’ ⟨*literal*⟩ {‘`,`’ ⟨*literal*⟩} ‘`)`’
| ⟨*attribute_path*⟩ ‘`.remove(`’ ⟨*literal*⟩ {‘`,`’ ⟨*literal*⟩} ‘`)`’

⟨*qualification*⟩ ::= ⟨*and_predicate*⟩ {‘ `or` ’ ⟨*and_predicate*⟩}

⟨*and_predicate*⟩ ::= ⟨*predicate*⟩ {‘ `and` ’ ⟨*predicate*⟩}

⟨*predicate*⟩ ::= ⟨*attribute_path*⟩ [‘`.`’ ⟨*aggregation*⟩] ⟨*compare_operator*⟩ ⟨*literal*⟩
| ⟨*attribute_path*⟩ [‘`.`’ ⟨*aggregation*⟩] ⟨*compare_operator*⟩ ⟨*attribute_path*⟩ [‘`.`’ ⟨*aggregation*⟩]
| ⟨*attribute_path*⟩ ‘ `is null`’
| ⟨*attribute_path*⟩ ‘ `is not null`’
| ‘`not` ’ ⟨*predicate*⟩
| ⟨*exist_predicate*⟩
| ⟨*forall_predicate*⟩
| ‘`(`’ ⟨*qualification*⟩ ‘`)`’

⟨*aggregation*⟩ ::= ‘`cnt()`’
| ‘`avg()`’
| ‘`min()`’
| ‘`max()`’
| ‘`sum()`’
| ‘`hmean()`’
| ‘`mode()`’

⟨*exist_predicate*⟩ ::= ‘`exist` ’ [‘`(`’ ⟨*integer*⟩ ‘`)` ’] ⟨*variable_name*⟩ ‘ `in` ’ ⟨*attribute_path*⟩ [‘`(`’ ⟨*qualification*⟩ ‘`)`’]

⟨*forall_predicate*⟩ ::= ‘`forall` ’ ⟨*variable_name*⟩ ‘ `in` ’ ⟨*attribute_path*⟩ ‘`(`’ ⟨*qualification*⟩ ‘`)`’

An *update* changes attribute values of entities or relationships. Attributes that should be updated must be listed in the *set* clause with their new values as constants.

Append can be used with multivalued attributes to add a value to the existing values: *emails.append('j.doe@doe.com', 'jdoe@doe.com')*. *Remove* can be used equivalently to delete values. To restrict the update on certain entities from the entity set or relationships from the relationship set, a *where* clause with a qualification can be added. The qualification predicates for monovalued attributes are basically identical to SQL. To express a predicate on a multivalued attribute, the user can either use one of the aggregation functions or the *exist* or *forall* predicates. We introduced them already in Section 4.6 with the selection operator. The number in the brackets after the *exist* keyword indicates how many distinct values must fulfill the inner predicate. If no number is provided, there must be at least one such value. A relationship update cannot change the participating entities but only the relationship's attribute values. If a participating entity should be changed, the old relationship needs to be deleted and a new one needs to be added because changing a participating entity would create a new relationship instance.

Listing 5.9: Updates of entity- and relationship sets

```
update Employee set ssn=987654320, name.fname='John', name.lname='Doe', salary=10000
    where ssn = 987654321;
update Employee set salary = 100000 where salary_history.avg()>100000;
update Department set locations.append('San Fransisco'), number=1, name='Research-A'
    where exist x in locations (x.city='Atlanta') and locations.cnt()=3;
update Department set locations.remove('San Fransisco'), number=1, name='Research-A'
    where (exist x in locations.city (x='Atlanta') and locations.cnt()=3) or
    locations is null;
update works_on set hours=25 where Employee=987654321 or Project=c('ProductX', 1);
```

5.2.3 Delete from

⟨*delete*⟩ ::= '**delete from** ' ⟨*name*⟩ [' **where** ' ⟨*qualification*⟩] '**;**'

A delete can remove entities and relationships and a where clause can be used to restrict the delete. If a strong entity is deleted, all weak entities that depend on that strong entity are also deleted.

Listing 5.10: Deletes from entity- and relationship sets

```
delete from Employee where salary>=100000 or salary_history.avg()>=100000;
delete from Department where forall x in locations.city (x='Atlanta' or x='Houston')
delete from supervision where Supervisee=987654321 or Supervisor=888665555;
delete from works_on where hours>=35 or hours<=5;
```

5.2.4 Select

The quality of a query language rises and falls with its flexibility for data retrieval. SQL has shortcomings here, because there are queries which are not expressible with SQL. Assume a relational database with a mapped schema from the conceptual schema in Fig. 4. The following queries are not expressible with a single SQL query: *'Give me all appointments, messages, and searched keywords of the user John Doe'* or *'Give me the number of messages of the user John Doe and the messages themselves'*. These queries only scratch the surface of SQL's shortcomings. There were proposals like SQL/NF [25] for a ¬1-NF relational model that tried to overcome these issues. However, if we use a ¬1-NF data model, why shouldn't we use the ER model directly? It offers much more semantics than any relational data model. With ERSQL, we tried to kill two birds with one stone: A more powerful data retrieval language that is comparable to and simple as SQL. Therefore, we export the semantic richness of the ER data model directly to the user through ERSQL.

⟨*select*⟩ ::= '`select` ' ⟨*select_clause*⟩
'`from` ' ⟨*from_clause*⟩
['`with` ' ⟨*with_clause*⟩]
['`reduce` ' ⟨*reduce_clause*⟩]
['`where` ' ⟨*qualification*⟩]
['`reduce` ' ⟨*reduce_clause*⟩]
['`collapse in` ' ⟨*name*⟩] ';'

The general *select* syntax is shown above. We introduce the different clauses in a step-by-step fashion now. We will also provide mappings for each clause that translate it into our algebra. All those mappings together form our canonical translation algorithm. We will illustrate the mappings with the following sample queries:

Listing 5.11: Query A: Get the number, the average salary, and the first and last names of all female employees.

```
select E.cnt() as enum, E.salary.avg() as avgsalary, E.name(fname, lname)
from Employee[sex = 'f']
collapse in E;
```

Listing 5.12: Query B: Get the ssn, last name, and the supervisor's last name (if there is one) as a composite attribute for all employees who work on all their projects more then 10 hours and work for the Research department. Further, get a list of their project names with the name of the department that controls the project.

```
select
     (ssn,name.lname,E2.name.lname as suplname) as einfo, P(P.name, D2.name as dname)
from Employee as E1,
     Employee as E2,
     Project as P,
     Department[name = 'Research'] as D1,
     Department as D2
with E1 as Supervisee +supervision E2 as Supervisor and
     E1 works_on P and
     E1 works_for D1 and
     P controls D2
where forall x in works_on (x.hours>10);
```

Listing 5.13: Query C: Get all pairs of employee last names and department names such that the employee works on at least one project, the department controls, for at least 10 hours. Further, only departments are considered which solely control projects that are located in the same cities the department has locations in.

```
select E.name.lname, D.name
from Employee as E,
     Department as D,
     Project as P
with E works_on P
     D controls P
reduce x in works_on (x.hours<10)
where works_on is not null and
    forall x in P.location (exist y in D.locations (x.city=y.city));
```

5.2.4.1 The FROM clause

⟨*from_clause*⟩ ::= ⟨*entity_type*⟩ [‘`[`’ ⟨*qualification*⟩ ‘`]`’] [‘ `as` ’ ⟨*alias_name*⟩]

The *from* clause in ERSQL defines, similar to SQL, the source entity types which are used to answer the query. Only entity types can be part of the *from* clause as our algebra is defined for entity types. We agree with Parent and Spaccapietra [23, 24] that entities model the real "things" in the mini-world and relationships are used to relate them (relationship joins). Furthermore, we extended our *from* clause by the so called *filters*. They can be added after each entity type in square brackets. They are useful to filter relevant entities early. They are also necessary because the *where* clause is applied after all relationship-joins, nested cartesian products, and merge joins were applied. Thus, attributes are often nested in multivalued attributes and then post filtering is difficult. We also find that filters make queries easier to read because filters are written next to the entity type they are applied on. Algorithm 1 defines how the *from* clause is mapped into our EER algebra. Fig. 17 shows the result of mapping the *from* clauses of our three sample queries with the mentioned algorithm.

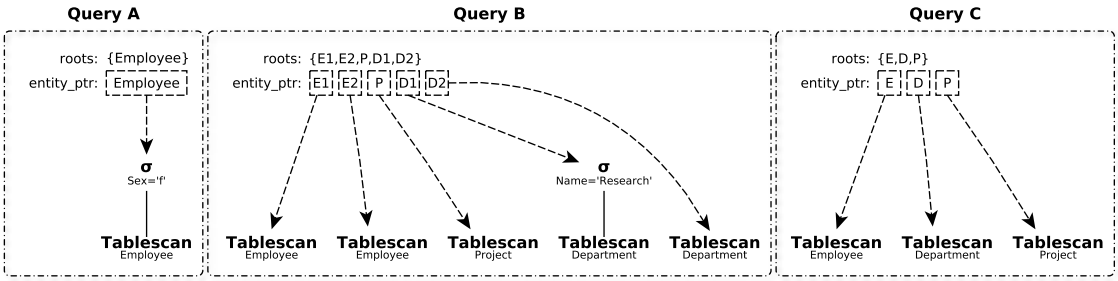


Figure 17: Sample queries after mapping *from* clause

Algorithm 1 Mapping the *from* clause

Global: Globally available data structures
set{*string*} *roots* : contains all roots of an algebra graph
map(*string* → *string*) *root_aliases* : for merge joins
map(*string* → *Operator**) *entity_ptr* : pointers to entity types
map(*string* → *string*) *merge_partner* : stores the current merge partner for an entity type

Input: *set* F : contains a triple (e, a, f) for each entry in *from* clause
e : entity type
a : alias (if defined)
f : filter predicate (if defined)

procedure MAP-FROM(F)
  for all $(e, a, f) \in F$ **do**
    $name \leftarrow e$
    if $a \neq \emptyset$ **then**
      $name \leftarrow a$
    end if
    $scan \leftarrow new\ Tablescan(e)$
    if $f \neq \emptyset$ **then**
      $filter \leftarrow new\ Selection(f)$
      ADDCHILD($parent : filter,\ child : scan$)
      $scan \leftarrow filter$
    end if
    $entity\_ptr[name] \leftarrow scan$
    $roots = roots \cup \{name\}$
    $root\_aliases[name] \leftarrow name$
  end for
end procedure

5.2.4.2 The WITH clause

⟨*with_clause*⟩ ::= ⟨*relation_definition*⟩ {‘ `and` ’ ⟨*relation_definition*⟩}

⟨*relation_definition*⟩ ::= ⟨*inner_relation_definition*⟩
  | ‘`(`’ ⟨*inner_relation_definition*⟩ ‘`) as` ’ ⟨*alias_name*⟩
  | ⟨*left_entity*⟩ ‘ `nest` ’ ⟨*right_entity*⟩

⟨*inner_relation_definition*⟩ ::= ⟨*left_entity*⟩ ‘ ’ [‘`+`’] ⟨*relationship_type*⟩ ‘ ’ ⟨*right_entity*⟩ {‘`,`’ ⟨*right_entity*⟩}

⟨*left_entity*⟩ ::= ⟨*entity_type*⟩ [‘ `as` ’ ⟨*rolename_name*⟩]

⟨*right_entity*⟩ ::= [‘`all` ’] ⟨*entity_type*⟩ [‘ `as` ’ ⟨*rolename_name*⟩]

The *with* clause links the entity types in the *from* clause by relationship joins and nested cartesian products. A merge join is applied between two entity types if they

join the same entity type(s) on the right side in prior relationship joins or nested cartesian products. To generate a nested cartesian product between an entity type E and F, with F as the right side of the product, "*E nest F*" must be added to the *with* clause. A relationship join of the entity type E with the entity types F, G through the relationship type *rel* can be expressed by an entry "*E rel F,G*". If an alias was defined for an entity type in the *from* clause, the alias has to be used. To use the outer join mode, a + must be added right in front of the relationship type name: *Employee +works_on Project* gets all employees with their projects but keeps employees that have no projects. To qualify a right side entity type with a $\forall$ quantifier, an *all* must be added before the entity type name: *Technican fixed Problem, all Client*. If the role names of the entity types in a relationship join can be inferred from the entity types themselves, they can be left out. The role name can be inferred for an entity type if the entity type participates only once in the corresponding relationship type. The entry *E1 as Supervisee supervision E2 as Supervisor* from query B illustrates a case for which role names are necessary to resolve ambiguity. Algorithm 2 defines how the *with* clause is mapped into our EER algebra. It defines the core part of the entire mapping algorithm because the result is an algebra operator graph with a single root for each query. We no longer mention an operator tree in our algebra because we allow our operators to have multiple parents. The main idea of the algorithm can be summarized in 4 steps:

1. Collect all entity types that occur on the left side of a relationship join or nested cartesian product.
2. For each of these entity types, generate all relationship join and nested cartesian products.
3. Check whether a merge join with other entity types is necessary due to the joins and products of step 2. Create the merge joins if necessary.
4. If there are still operator graphs, that are not linked, link them with a cartesian product.

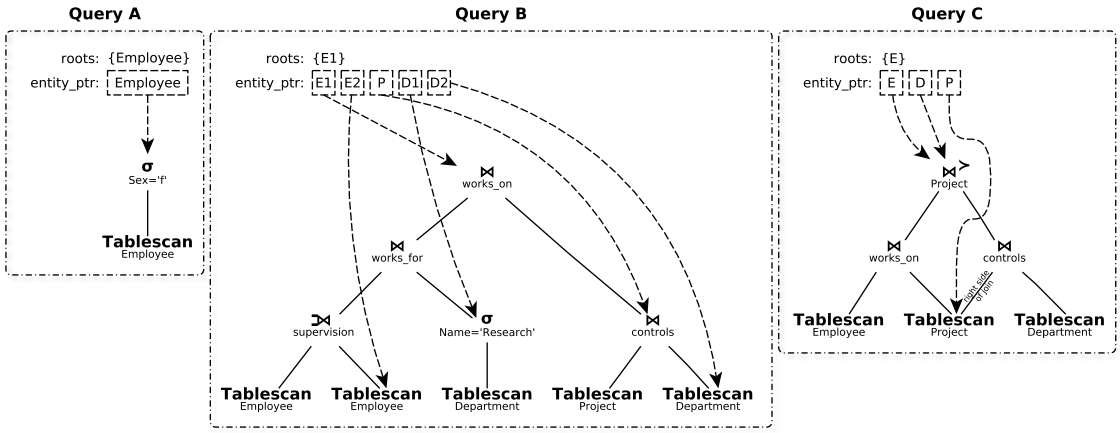


Figure 18: Sample queries after mapping *with* clause

The operators in step 2 can be applied in an arbitrary order for each left side entity type. Finding the most efficient order will be the topic to our future work on ERSQL query optimization. Fig 2 illustrates one possible result after mapping the with clause for our three sample queries.

5.2.4.3 The WHERE clause

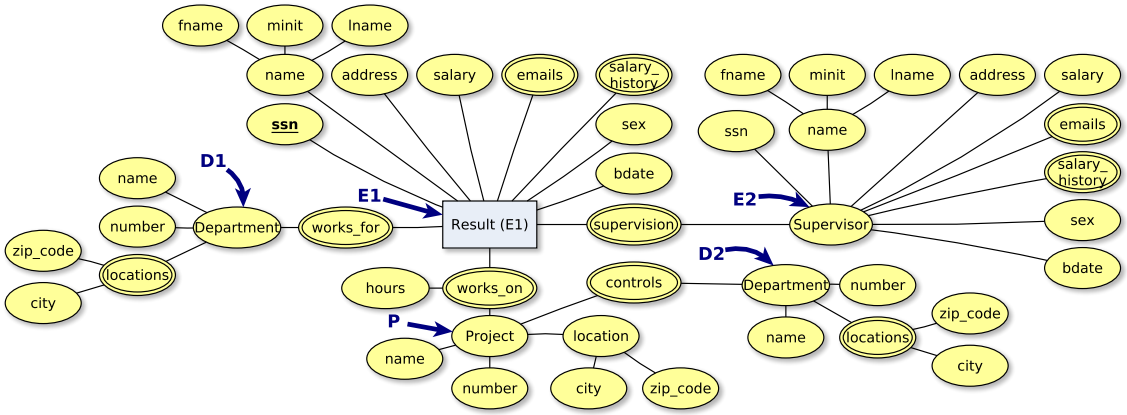


Figure 19: Entity set after applying *with* clause of sample query B with mnemonics (blue arrows)

The *where* clause filters entities from the entity set that is created by the with clause. Thus, the qualification must be expressed on the corresponding entity type

Algorithm 2 Mapping the *with* clause

Input: *set* R : contains a quadruple (e_l, r_l, r, p) for each relationship join
e_l : entity type of the left join partner
r_l : role name of the left join partner
r : relationship type
p : contains a triple (e_r, r_r, q_r) for every right join partner
e_r : entity type of the right join partner
r_r : role name of the right join partner
q_r : quantifier of the right join partner ($\exists$ or $\forall$)
set P : contains a tuple (e_l, e_r) for each nested cartesian product
e_l : left entity type
e_r : right entity type

procedure MAP-WITH(R, P)
  $left \leftarrow \emptyset$
  for all $(e_l, r_l, r, p) \in R$ **do**
    $left = left \cup \{e_l\}$
  end for
  for all $(e_l, e_r) \in P$ **do**
    $left = left \cup \{e_l\}$
  end for
  for all $e \in left$ **do**
    $merge\_map \leftarrow \emptyset$ ▷ is given to procedures by reference
    $entity \leftarrow entity\_ptr[e]$
    CREATERELATIONSHIPJOINS($e, R, merge\_map$)
    CREATENESTEDCARTPRODUCTS($e, P, merge\_map$)
    $left\_is\_root \leftarrow$ CREATEMERGEJOINS($e, merge\_map$)
    $entity_{new} \leftarrow entity\_ptr[e]$
    if $left\_is\_root \neq true$ **then**
      $roots \leftarrow roots - \{e\}$
    end if
    for all $p \in$ GETPARENTS($entity$) **do**
      REPLACECHILD($parent : p, child : entity, child_{new} : entity_{new}$)
    end for
    $entity\_ptr[e] \leftarrow left$
    for all $(e_m, m) \in merge\_map$ **do**
      $entity_m \leftarrow entity\_ptr[e_m]$
      for all $p \in$ GETPARENTS($entity$) **do**
        REPLACECHILD($parent : p, child : entity_m, child_{new} : entity_{new}$)
      end for
      $entity\_ptr[e_m] \leftarrow entity_{new}$
    end for
  end for
  return CREATECARTESIANPRODUCTS()
end procedure

```
procedure createRelationshipJoins(e, R, merge_map)
    left ← entity_ptr[e]
    for all (e_l, r_l, r, p) ∈ R with e_l = e do
        join ← new RelationshipJoin(e_l, r_l, r, p)
        AddLeftChild(parent : join, child : left)
        for all (e_r, r_r, q_r) ∈ p do
            right ← entity_ptr[e_r]
            AddRightChild(parent : join, child : right)
            if merge_partner[e_r] ≠ ∅ then
                e_m ← merge_partner[e_r]
                merge_map[e_m] = merge_map[e_m] ∪ {e_r}
            end if
            merge_partner[e_r] ← e
            roots ← roots − {e_r}
        end for
        left ← join
    end for
    entity_ptr[e] ← left
end procedure

procedure createNestedCartProducts(e, P, merge_map)
    left ← entity_ptr[e]
    for all (e_l, e_r) ∈ P with e_l = e do
        right ← entity_ptr[e_r]
        cartesian = new NestedCartesianProduct()
        AddLeftChild(parent : cartesian, child : left)
        AddRightChild(parent : cartesian, child : right)
        if merge_partner[e_r] ≠ ∅ then
            e_m ← merge_partner[e_r]
            merge_map[e_m] = merge_map[e_m] ∪ {e_r}
        end if
        merge_partner[e_r] ← e
        roots ← roots − {e_r}
        left ← join
    end for
    entity_ptr[e] ← left
end procedure
```

```
procedure createMergeJoins(e, merge_map)
    left ← entity_ptr[e]
    left_is_root ← true
    for all (e_m, m) ∈ merge_map do
        right ← entity_ptr[e_m]
        merge = new MergeJoin(m)
        AddLeftChild(parent : merge, child : left)
        AddRightChild(parent : merge, child : right)
        left_is_root ← left_is_root & (e_m ∈ roots)
        roots ← roots − {e_m}
        root_aliases[e_m] ← root_aliases[e]
        for all (e_x, a) ∈ root_aliases do
            if a = e_m then
                root_aliases[e_x] ← root_aliases[e]
            end if
        end for
        left ← merge
    end for
    entity_ptr[e] ← left
    return left_is_root
end procedure

procedure createCartesianProducts()
    root ← ∅
    for all e ∈ roots do
        if root = ∅ then
            root ← entity_ptr[e]
        else
            right ← entity_ptr[e]
            cartesian ← new CartesianProduct()
            AddLeftChild(parent : cartesian, child : root)
            AddRightChild(parent : cartesian, child : right)
            root ← cartesian
        end if
        roots = roots − {e}
    end for
    return root
end procedure
```

(created by *with* clause). Fig. 19 shows the entity type of our sample query B after the *with* clause was applied. The attribute paths in such an entity type can become long. That is why we introduced so called *mnemonics*. Mnemonics are short cuts to access certain attributes without specifying the whole attribute path. They are added for each entity type in the *from* clause and point to the corresponding, possibly nested entity type directly. If an alias is defined in the *from* clause, the alias is used for the mnemonic. In Fig. 19, $D2.name$ is equivalent to $works\_on.Project.controls.Department.name$. Another syntactic feature is the implicit existence predicate. Whenever an attribute path contains too many levels of multivalued attributes for a certain predicate, implicit existence predicates are added. Take the predicate *D1.name = 'Research'* as an example. The path $D1.name$ has one multivalued attribute ($works\_for$) but the predicate is a monovalued predicate. Internally, an existence quantifier is added: *exist x in D1.name (x = 'Research')*. Another example is the predicate *exist x in D2.name (x = 'Research')*. The attribute path contains 2 levels of multivalued attributes but the exist predicate only works for attribute paths with one level of multivalued attributes. Thus, it is extended to *exist y in works_on.Project (exist x in y.controls.Department.name (x = 'Research'))*. The *where* clause is mapped into our algebra by adding a selection on top of the algebra graph.

5.2.4.4 The REDUCE clause

⟨*reduce_clause*⟩ ::= ⟨*mv_reduction*⟩ {‘ `and` ’ ⟨*mv_reduction*⟩}

⟨*mv_reduction*⟩ ::= ⟨*variable_name*⟩ ‘ `in` ’ ⟨*attribute_path*⟩ ‘ `(`’ ⟨*qualification*⟩ ‘`)`’

The *reduce* clause(s) limit the population of a multivalued attribute based on a reduction predicate. The reduction syntax for a multivalued attribute is similar to an existence or forall predicate. First, a bind variable is defined which iterates over the values of the multivalued attribute during evaluation. The reduction predicate can be defined in brackets after specifying the attribute path to the multivalued attribute

that should be reduced. The bind variable can be used within the reduction predicate. If the predicate evaluates to true for the current value the bind variable carries, the value is omitted. The *reduce* clause(s) can be mapped into our algebra by adding a reduction operator on top of the algebra graph. There are two places for *reduce* clauses: before and/or after the *where* clause. The reductions before the *where* clause are applied before the selection operator of the where clause. The other reductions are applied after it. Listing 5.14 shows some *reduce* clauses for the resulting entity set of query B (see Fig. 19).

Listing 5.14: Sample *reduce* clauses

```
reduce x in works_on (x.hours<10) and x in supervision (x.Supervisor.sex='m')
reduce x in emails (x='john.doe@company.com')
```

5.2.4.5 The COLLAPSE clause

The *collpase* clause collapses the entities in the resulting entity set into a single entity. The name of the new multivalued collapse attribute can be specified within the clause. The clause can be mapped into our algebra by adding a collapse operator on top of the algebra graph.

5.2.4.6 The SELECT clause

⟨*select_clause*⟩ ::= ⟨*attribute_transformation*⟩ {‘,’ ⟨*attribute_transformation*⟩}
| ‘*’

⟨*attribute_transformation*⟩ ::= ⟨*inner_attribute_transformation*⟩ [‘ as ’ ⟨*new_name*⟩]

⟨*inner_attribute_transformation*⟩ ::= ⟨*attribute_path*⟩ [‘.’ ⟨*aggregation*⟩]
| ‘(’ ⟨*select_clause*⟩ ‘)’
| ‘{’ ⟨*inner_attribute_transformation*⟩ ‘}’
| ⟨*attribute_path*⟩ ‘(’ ⟨*select_clause*⟩ ‘)’

The *select* clause can be used to transform the resulting entity type. An asterisk indicates that no transformation should be applied. Otherwise, a comma-seperated list of *attribute transformation descriptions* has to be given. One description per attribute that the final entity type should have. If an attribute should be renamed,

a new name can be provided directly after the description with: *as <new_name >*. There are four types of attribute transformation descriptions: *attribute path*, *composite creation*, *multivalue creation*, and *composite access*. The use of mnemonics is allowed in all of them.

Attribute paths can be used to select an existing attribute or to project a (multivalued) composite attribute onto a component. Further, they can be used to return the value of an aggregation function. The query in Listing 5.15 returns the ssn, first name, and the number of emails for each employee:

Listing 5.15: Attribute paths in *select*

```
select ssn, name.fname as firstname, emails.cnt() as emailnum from Employee
```

To generate a new composite attribute, a *composite creation* description can be used. In between brackets, a comma-seperated list of attribute transformation descriptions has to be provided. One description for every component of the new composite attribute. If no name is provided for the new composite, the database system has to name it.

Listing 5.16: Composite creation in *select*

```
select (ssn, name.lname, emails.cnt() as emailnum) as einfo from Employee
```

A *multivalue creation* description creates a new multivalued attribute. In between curly brackets, an attribute transformation description for a monovalued attribute has to be provided. This attribute is transformed then into a multivalued attribute. If no new name is provided, the name of the inner attribute is used.

Listing 5.17: Multivalue creation in *select*

```
select {ssn}, {salary} as mv_salary, {(ssn, name.lname)} as einfo from Employee
```

Composite access descriptions project a (multivalued) composite onto a subset of its components. But it is also possible to add new components to an existing composite attribute. A *composite access* description starts with an attribute path to a

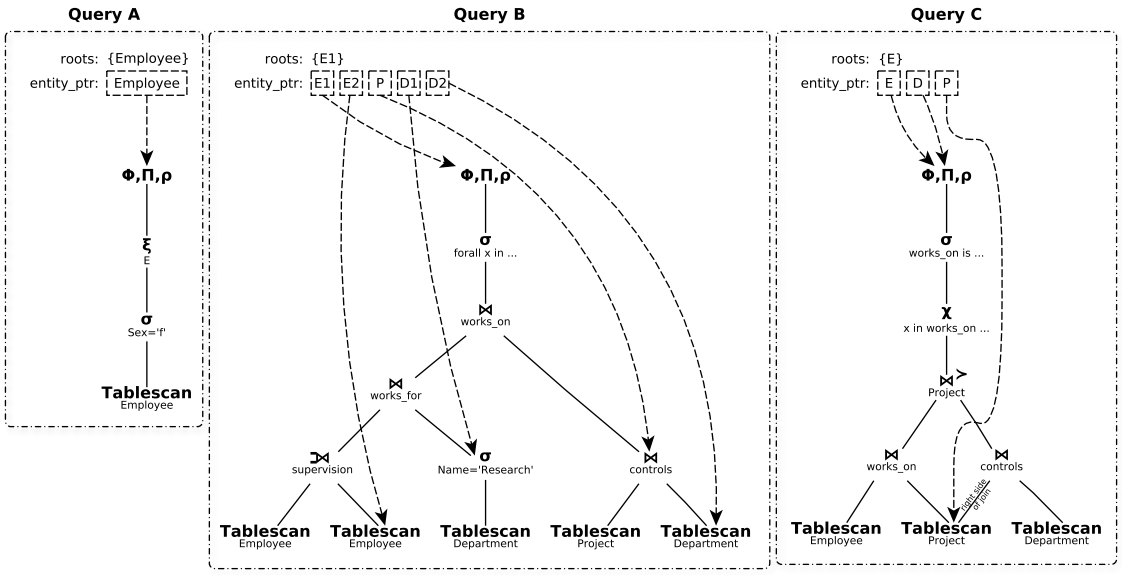


Figure 20: Sample queries after complete mapping

composite attribute followed by a list of comma-seperated attribute transformation descriptions in brackets. The list contains one description per component. The original components of the composite attribute can be accessed without specifying the path to the composite attribute first: e.g. *name(fname, lname)* in *Employee*. If the source entity type already has an attribute with the same name as a component, the whole path must be used for the component. Sample query B shows such a case: *P(P.name, D2.name...)*.

Listing 5.18: Composite access in *select*

```
select location(name, zip_code, city) as address from Project
```

To map the select clause into our EER algebra, we combined the *projection*, *casting*, and *renaming* operator into a single operator. Otherwise, we would have to apply multiple rounds of the operators due to the recursive definition of our attribute transformation descriptions. That recursion allows us to build arbitrary hierarchical output formats. Fig. 20 shows the complete translation of our sample queries into our EER algebra.

5.2.5 Union, intersect, and except

⟨*union*⟩ ::= ⟨*retrieval_query*⟩ ' `union` ' ⟨*retrieval_query*⟩ ';'

⟨*intersect*⟩ ::= ⟨*retrieval_query*⟩ ' `intersect` ' ⟨*retrieval_query*⟩ ';'

⟨*except*⟩ ::= ⟨*retrieval_query*⟩ ' `except` ' ⟨*retrieval_query*⟩ ';'

⟨*retrieval_query*⟩ ::= ⟨*inner_retrieval_query*⟩
| '(' ⟨*inner_retrieval_query*⟩ ')'

⟨*inner_retrieval_query*⟩ ::= ⟨*select*⟩
| ⟨*union*⟩
| ⟨*intersect*⟩
| ⟨*except*⟩

As in SQL, *union*, *intersect*, and *except* combine two subqueries by a set operation. They can be mapped by combining the algebra expressions of both subqueries with an union, intersection, or respectively difference operator. Listing 5.19 gives some sample queries with set operations. The first query returns the name, number, and locations of all departments and projects. The second query returns all salaried managers that are also engineers. And query three returns all employees that are neither hourly employees nor technicans. As our EER algebra is set-based, possible duplicates are omitted for union queries.

Listing 5.19: Sample set queries in ERSQL

```
select name, number, locations from Department
union
select name, number, {location} from Project;

select ssn from SalariedManager
intersect
select ssn from Engineer;

select * from Employee
except
(select * from HourlyEmployee
    union
 select * from Technican)
```

5.2.6 Un-nesting subqueries into the main query

In SQL, subqueries are often used to overcome issues related to the flatness of the relational model. Typical queries that require a subquery are: *Give me the ssn of all employees that earn more than the average* or *Give me the ssn of all technicans that have fixed all problems.* Listing 5.20 shows possible ways to answer these queries in SQL and in ERSQL.

Listing 5.20: Same queries in SQL and ERSQL

```
-- SQL:
select ssn from Employee
where salary > (select avg(salary) from Employee);

select ssn from Technican as T
where not exists (select * from Problem as P
                  where not exists (select * from fixed
                                    where technican = T.ssn and problem = P.id));

-- ERSQL:
select ssn from Employee as E, Employee as E2 with E nest E2
where E.salary > E2.salary.avg();

select ssn from Technican as T, Problem as P with T fixed all P;
```

In both cases, ERSQL needs no subquery. For the first query, we use a technique we call *nesting*. We incorporate the subquery into the main query via a nested cartesian product. The second query needs no subquery due to the CISC operators of our EER algebra. Even for more advanced queries like *Give me the ssn of all employees of the Administration department who earn more than the top earner of the Research department* or *Give me the ssn of all employees who work on all projects controlled by the Research department*, ERSQL needs no subqueries:

Listing 5.21: Advanced ERSQL queries

```
select ssn
from Employee as E1,
     Department[name='Administration'] as D1,
```

```
      Employee as E2,
      Department[name='Research'] as D2
with E1 works_for D1 and
      E2 works_for D2 and
      E1 nest E2
where E1.salary > E2.salary.max();

select ssn
from Employee as E,
      Project as P,
      Department[name='Research'] as D
with E works_on all P and
      P controls D;
```

We will investigate whether there is a useful class of queries that would require subqueries in ERSQL or whether all reasonable queries can be covered by the presented *nesting* technique and the CISC operators of our algebra.

5.3 Examples of ERSQL queries

The following queries should give a good idea about how ERSQL can be used to answer queries (based on schema in Fig. 8):

Q1: *Get all employees with their supervisor, if they have one, their department, their projects, and their dependents, if they have any:*

```
select *
from
    Employee as E1,
    Employee as E2,
    Department as D,
    Project as P,
    Dependent as De
with
    E1 as Supervisee +supervision E2 as Supervisor and
    E1 works_for D and
    E1 works_on P and
    E1 +dependents_of De;
```

Q2: *Get the name of projects which have at least one employee from the 'Research' department working on it:*

```
select name
from
    Project as P,
    Employee as E,
    Department[name='Research'] as D
with
    P works_on E and
    E works_for D;
```

Q3: *Get the number of managers, their average salary, and their last names:*

```
select M.cnt() as mnum, M.salary.avg() as msal, M.name.lname
from Manager
collapse in M;
```

Q4: *Get the first and last names of employees who work for the 'Research' department and work on the 'Reorganization' project:*

```
select name.fname, name.lname
from
    Employee as E,
    Project[name='Reorganization'] as P,
    Department[name='Research'] as D
with
    E works_on P and
    E works_for D;
```

Q5: *Get the first and last names of employees who work for the 'Research' department and/or work on the 'Reorganization' project:*

```
select name.fname, name.lname
from
    Employee as E,
    Project[name='Reorganization'] as P,
    Department[name='Research'] as D
with
    E +works_on P and
    E +works_for D
where not (P is null and D is null);
```

Q6: *Get the ssn and name of all employees, who earn more than the top earner of all departments except their own, as a composite attribute 'einfo':*

```
select (ssn, name) as einfo
from
    Employee as E1,
    Employee as E2,
    Department as D1,
    Department as D2
with
    E1 works_for D1 and
    D2 works_for E2 and
    E1 nest D2
where forall x in D2 (exist y in D1 (x.name=y.name) or E2.salary.max()<E1.salary);
```

Q7: *Get the name of all departments that only control projects which are located in cities the department has also locations in:*

```
select name
from
    Department as D,
    Project as P
with D controls P
where forall x in P.location.city (exist y in locations.city (x=y));
```

Q8: *Get all pairs of employees that work for the same department and work on at least one project together:*

```
select *
from
    Employee as E1,
    Employee as E2,
    Department as D,
    Project as P
with
    E1 works_for D and
    E1 works_on P and
    E2 works_for D and
    E2 works_on D
where E1.ssn <> E2.ssn;
```

Q9: *Get all employees' ssn and last name who work on at least 3 of their projects for more than 10 hours:*

```
select ssn, name.lname
from
    Employee as E,
    Project as P
with E works_on P
where exist(3) x in works_on (x.hours>10);
```

Q10: *Get all employees' ssn and last name who work on at least 5 projects and who work on all of their projects for more than 10 hours:*

```
select ssn, name.lname
from
    Employee as E,
    Project as P
with E works_on P
where P.cnt()>=5 and forall x in works_on (x.hours>10);
```

CHAPTER VI

THE ERDBMS PROTOTYPE

We now provide some implementation details briefly on how we store an ER schema and data physically and on how we implement the CISC operators in an efficient fashion. As main-memory capacities have increased considerably and high end servers offer main-memory in magnitudes of TB, we decided to implement our *ERDBMS* prototype system as a pure main-memory DBMS. This approach has been followed in the research system HyPer [15] and the popular commercial system SAP Hana [10]. These systems offer high performance query processing because they discard all overhead related to disk storage. Harizopoulos et al. [12] have shown that almost 35% of the query processing time in a traditional DBMS is spent for buffering of disk pages. A high performance query engine is necessary to make our CISC operators implementable. We achieve durability by logical logging which is possible because no inconsistent state is written out to disk. Furthermore, our prototype system exploits the semantics, available in the ER data model, to automatically generate indexes for efficient query processing. The system also uses these indexes to keep track of inconsistencies due to relationship constraint violations or specialization constraint violations.

6.1 Data storage layer and automatic index generation

The virtual address space of a process is a continuous array of storage cells. Thus, storing an ER schema in an efficient fashion is a challenging task because it is not flat (as we heavily use multivalued attributes). On the other hand, storing relations from the relational model is a lot easier because of its flat characteristic. The rows of a relation can either be stored in a vector of tuples (row store) or in multiple vectors where each vector stores the values for one column (column store). There are efficient standard implementations for vectors and tuples in almost every programming language. Thus, the relational model can be considered a physical data model when main memory is used for storage. We can physically store an ER schema now

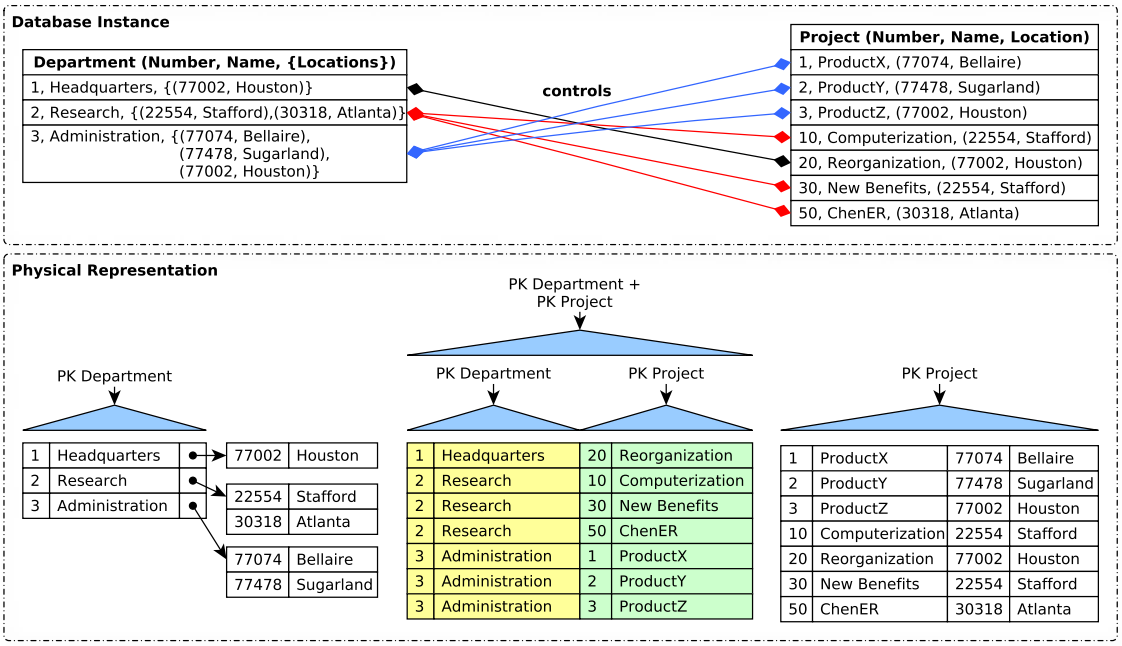


Figure 21: Storage of *Project* and *Department* entity sets and the *controls* relationship set (string values are from fixed length *char* value set) and automatically maintained indices (blue)

by using the well-defined mapping techniques into the relational model [6, 8]. Fig. 21 illustrates how the entities in *Department* and *Project*, and the *controls* relationships among them are physically stored (row-store representation, but column store is also possible). Each entity type is stored in its own vector of entity tuples. For a multivalued attribute, a vector with the current values is generated for each entity and a pointer to it is stored in the original entity tuple. If there are multiple levels of multivalued attributes, such an indirection (through a pointer) is added for each level. Values for a varchar attribute are stored consecutively in a seperately allocated buffer and a pointer to the first character of the string is stored in the entity tuple. As deletes and updates can lead to holes in a varchar buffer, periodical compressions and pointer adjustments have to take place. Weak entity types are stored by adding the primary key of all defining entities to the stored weak entity tuple. Relationships are represented by storing a tuple with all primary keys of the participating entities

and the values for the relationship type attributes. We store the primary keys instead of a pointer to the entities because the position of an entity in the entity type vector might change over its lifetime. Specializations can be stored in multiple ways (see [8] for alternatives). We store a bitfield with each entity tuple. The bitfield contains as many bits as there are subclasses for an entity. For the *Employee* entity type, a 7 bit field is stored with each entity. A distinct index between 0 and 6 is assigned to each subclass of *Employee*. A 1 at index i in the bitfield of an *Employee* entity e indicates, that e is member of the subclass with index i. If an entity is added to a subclass, the bits with indices of superclasses have to be set as well. If an entity is deleted from a superclass, the bits with indices from subclasses have to be set to 0. An entity is physically deleted if it is deleted from the root of the specialization hierarchy. Total vs. partial specialization constraints are incorporated using the bitfields. The ER model provides knowledge about which entity types will be joined using relationship types with which other entity types and thus automatic index generation is possible. Our system automatically generates an index for a primary key lookup for each entity type. Furthermore, it generates for each relationship type an index for the primary key of each participating entity type. Relationship joins and min-max constraint checking can be implemented efficiently with them. An index for the composition of all participating primary keys is also maintained which can be used to check whether a relationship already exists.

6.2 CISC operators with data-centric code generation

Listing 6.1: Data-centric code generation (example from [20])

```
scan - operator:
    scan.produce()
        print "for each" + entity + " in entity set"
            scan.parent.consume(entity, this)
σ selection - operator:
    σ.produce()
```

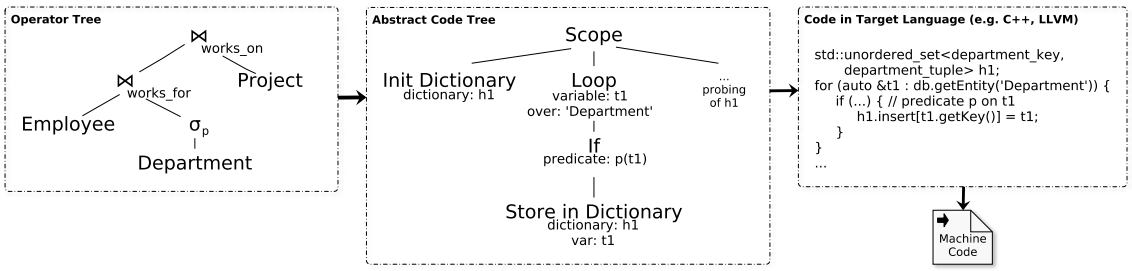


Figure 22: Different intermediary artefacts during query processing

```
    σ.child.produce()
σ.consume(entity, src)
    print "if " + σ.predicate
        σ.parent.consume(entity, this)
```

Our algebra operators are more complex than the operators in the relational algebra. Therefore, their efficient implementation has a lot of potential for improvement. Neumann showed that traditional iterator models perform poorly in a main-memory only environment and thus he proposed his data-centric code generation approach [19, 20]. Data-centric code generation is efficient because the data tuples are no longer moved between operator boundaries but the operators are applied in a pipeline fashion on the same data item. Except for pipeline breaker operators, no intermediary copies are necessary. This results in good caching behavior and leads to less instruction mispredictions by the CPU. We adapted Neumann's model slightly, by introducing another abstraction layer called an *abstract code tree (ACT)*. Fig. 22 shows the different intermediary artifacts during the processing of an algebra expression. In step 1, an operator tree is translated into an ACT. Therefore, an operator has to implement the two methods *produce* and *consume*. When *produce* is called on an operator, it must generate the ACT description of its implementation. It can force its childs to produce their ACT via invoking *produce* on them or it can let its parent consume a tuple by calling *consume*. Listing 6.2 illustrates how the scan and selection operator can be implemented with Neumann's model. *Print* means that

the ACT for the described action is created there. For more information, we refer the reader to Neumann's work. The resulting ACT of the first transformation step is basically an abstract algorithmic description of the query processing. We introduced the ACT to separate the code generation from the operator implementation. Furthermore, we believe that the finer grained ACT offers optimization potential for situations, that cannot be generalized for the operator tree level. Investigating such ACT optimization techniques will be, beside operator tree optimization techniques, a topic to future work. In a third transformation step an ACT is translated into a compileable target language such as C++ or LLVM code. Currently, we use C++ which shows efficient execution times but poor compilation times. We will face this issue by translating the ACT into a lower level intermediary representation language like LLVM code which can be compiled faster. We already allow stored, precompiled queries. They are queries which contain placeholders. Whenever a precompiled query is to be executed, concrete values for the placeholders have to be provided. Listing 6.2 illustrates how one can use precompiled queries. Executing a precompiled query does not require any compilation as the query machine code translation is already available. All in all, the implementation of all our algebra operators has been straightforward with Neumann's operator model. In our current implementation we are able to map the ERSQL queries into the proposed algebra presented in this paper and for reasonable size data, the performance is satisfactory. We are yet to conduct detailed performance studies.

Listing 6.2: Prepared statement example

```
prepare select * from Employee as E, Project as P
        with E works_on P
        where name.lame=@lname;
-- database system returns unique id for prepared query, e.g. 1234
execute 1234 with @lname='Doe';
execute 1234 with @lname='Borg';
```

6.3 Efficient semantic analysis

Another challenging task during the development of our prototype system was an efficient implementation of the semantic analysis for the *where*, *reduce*, and *select* clauses. In SQL, the semantic analysis for these clauses is straightforward due to the flatness of the relational model. The ER model, as we defined it, on the other hand allows nested multivalued and composite attributes. The mentioned clauses operate on the entity type that is created by the *with* clause. We need a description of that entity type to perform a semantic analysis. Therefore, we extended Neumann's operator model to provide us with the description. Beside *produce* and *consume*, each operator also implements a *resultEntitySetDescription* method. When called, the method has to return the description of the entity type after the corresponding operator was applied. To create this description, the operator can use the entity type descriptions of its childrens if necessary. An entity type description contains a list of all entity type attributes. Furthermore, it must contain a description of the domain in case of a multivalued attribute and a description of all component attributes in case of a composite attribute. With the help of such a description, the semantic analysis of the mentioned clauses of our ERSQL query can be performed efficiently. The approach can be described in the following 5 Steps:

1. Perform a semantic analysis of the *from* clause - are all entity types valid?

2. Perform a semantic analysis of the *with* clause - are all relationship joins and nested cartesian products valid?

3. Map the *from* and *with* clause into an ERR-algebra expression as described in Section 5.2.4.

4. Get the entity set description with the help of the created operator graph.

5. Perform the semantic analysis for the *reduce*, *where*, and *select* clauses (if there

is a collapse operator, the description after collapsing must be used for the select).

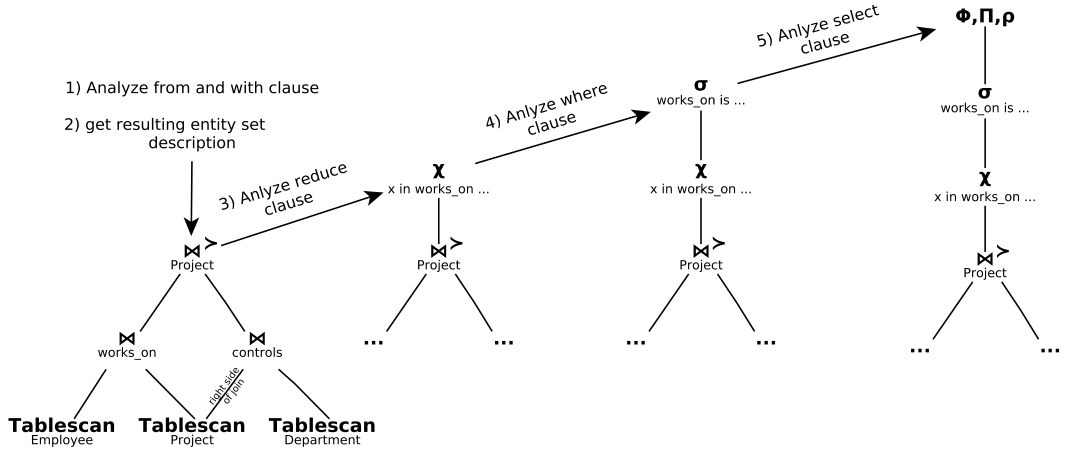


Figure 23: Semantic analysis for sample query C from section 5.2.4

CHAPTER VII

FUTURE WORK & CONCLUSIONS

This work presented our proposal to treat the ER model as a logical data model in which users can write queries and build applications. We defined an EER algebra, which is heavily based on the work of Parent and Spaccapietra [23, 24], and our query language ERSQL. The main design goals of ERSQL were twofold: keep it as simple as SQL while exploiting the powerfulness of the ER model. The ER model is a more abstract data model than the relational model because it allows a non-1-NF schema to be represented and models the mini-world with entities and their interrelationships. This leads to a model of the mini-world that is close to the natural perception of human beings. Exposing this semantic rich data model directly to the user through a high level query language like ERSQL results in queries that are closer to natural language. Another point, ERSQL benefits of, is our EER algebra with its CISC operators. We extended the outstanding algebra proposal by Parent and Spaccapietra [23, 24] with quantifiers for the relationship join and some additional operators to have a sound basis for our ERSQL language. Especially, the forall quantifiers for the relationship join simplify a large class of queries that need to be expressed with double negation in SQL. Our prototype system shows, that all CISC operators can be implemented efficiently, when modern implementation techniques are used: Exploiting main-memory capacities and compiling queries into machine code. Neumann's data centric code compilation approach [19, 20] was essential to clear the way to an efficient implementation of our CISC operators.

There is still a great deal of research necessary to make an ERDBMS deployable in a productive environment. The ER model allows to express extensive constraints over the database state (e.g. relationship constraints or specialization constraints). Our prototype system already keeps track of constraint violations in an almost no overhead manner. However, in a productive system, it is necessary that the constraints are enforced over the database state at any point in time. Therefore, we plan to introduce a tailored transaction concept for the ER model. It should allow

temporal violations of database constraints within a transaction. However, a transaction can only commit successfully if all temporal constraint violations have been fixed. Take the database schema in Fig. 8 as an example: One cannot add an employee without violating some constraints (e.g. relationship constraint on *works_for* relationship type or participation constraint on employee specialization). Within the same transaction these constraints must be fixed by creating, for example, a relationship for the employee in *works_for* and adding the employee to one of the subclasses for the total employee specialization (*HourlyEmployee* or *SalariedEmployee*). If all constraint violations have been fixed, the transaction can commit successfully. After finishing the implementation of a transaction model, we will be able to conduct performance benchmarks. We think that an ERDBMS has two advantages over a standard DBMS: First, our CISC operators can perform more work with less operators. Second, ERSQL allows to combine multiple SQL queries into a single ERSQL query.

Another part of future work will be optimization techniques for our EER algebra. In contrast to the relational algebra, our operators might have multiple parents. The result calculated by the child must not be recalculated for every parent operator. Therefore, we plan to introduce a temporary storage operator. Moreover, join order optimization is also an important topic. In contrast to joins in relational systems, the relationship joins, nested cartesian products, and merge joins cannot be arbitrarily reordered because they are not commutative nor associative. We think this might even be an advantage as it reduces the search space for the optimal join order. Evaluating the optimization potential and identifying optimization techniques for our ACTs will also be part of future work.

While ERSQL already offers a powerful query interface, it still lacks two important concepts: sorting and attribute grouping. In contrast to a SQL query, we can have nested entity sets in multivalued attributes. Therefore, multiple dimensions for

sorting exist. We are currently working on a simple solution to specify sorting for different dimensions. Furthermore, in SQL, grouping by a relation attribute is often used to build groups within a relation (e.g. grouping employees into males and females). We plan to incorporate such grouping techniques with multivalued attributes in ERSQL.

APPENDIX A

ERSQL

A.1 Data definition language - Example

Listing A.1: Schema definition of example schema in Fig. 8

```
create type EMailType from Varchar(150) (
    format '^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+.[a-zA-Z]{2,4}'
);
create type SalaryType from Integer (
    range min(10000) max(100000)
);
create type GenderType from Char(1) (
    format '^(m|f)'
);
create entity Employee (
    ssn Integer,
    bdate Date,
    emails Multivalued ( EMailType ) not null,
    name Composite (
        fname Varchar(50) not null,
        minit Char(1),
        lname Varchar(50) not null
    ),
    address Varchar(150) not null,
    salary SalaryType not null default 30000,
    salary_history Multivalued ( SalaryType ),
    sex GenderType not null,
    key (ssn)
);
create entity Department (
    name Varchar(50),
    number Integer,
    locations Multivalued ( Composite (
        zip_code Integer not null,
        city Varchar(100) not null
```

```
    ) ) not null,
    key (name, number)
);
create entity Project (
    name Varchar(50),
    number Integer,
    location Composite (
        zip_code Integer not null,
        city Varchar(100) not null
    ),
    key (name, number)
);
create entity Client (
    ssn Integer,
    name Composite (
        fname Varchar(50) not null,
        minit Char(1),
        lname Varchar(50) not null
    ),
    address Varchar(150) not null,
    emails EMailType,
    key (ssn)
);
create entity Problem (
    id Integer,
    type Varchar(50) not null default 'maintenance',
    key (id)
);
create weak entity Dependent (
    defined by (
        create relationship Dependent_of (
            entities (
                Employee min(1)
            )
        )
    ),
    name Varchar(50) not null,
    bdate Date,
    sex GenderType,
    address Varchar(150) not null,
    partial key (name)
```

```
);
create partial distinct specialization of Employee (
    create entity Secretary (
        typing_speed Integer not null
    ),
    create entity Technican (),
    create entity Engineer ()
);
create partial distinct specialization of Employee (
    create entity Manager ()
);
create total distinct specialization of Employee (
    create entity HourlyEmployee (
        working_hours integer default 10
    ),
    create entity SalariedEmployee ( )
);
create partial distinct specialization of Manager (
    create entity SalariedManager ()
);
create partial distinct specialization of SalariedEmployee (
    create entity SalariedManager ()
);
create relationship supervision (
    entities (
        Employee Supervisee max(1),
        Employee Supervisor
    )
);
create relationship works_for (
    entities (
        Employee min(1) max(1),
        Department min(5)
    )
);
create relationship works_on (
    entities (
        Employee min(1) max(10),
        Project min(3)
    ),
    attributes (
```

```
        hours Integer not null default 5
    )
);
create relationship controls (
    entities (
        Department,
        Project min(1) max(1)
    )
);
create relationship manages (
    entities (
        SalariedManager max(1),
        Department min(1) max(1)
    ),
    attributes (
        start_date Date
    )
);
create relationship fixed (
    entities (
        Technican,
        Client min(1),
        Problem min(1)
    )
);
```

REFERENCES

[1] Androutsopoulos, I., Ritchie, G., and Thanisch, P., "MASQUE/SQL: An Efficient and Portable Natural Language Query Interface for Relational Databases," in *Proceedings of the 6th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, IEA/AIE'93, pp. 327–330, Gordon & Breach Science Publishers, 1993.

[2] Auxerre, P. and Inder, R., "MASQUE Modular Answering System for Queries in English-User's Manual," tech. rep., Technical Report AIAI/SR/10, Artificial Intelligence Applications Institute, University of Edinburgh, 1986.

[3] Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P., "A critique of ANSI SQL isolation levels," in *ACM SIGMOD Record*, vol. 24, pp. 1–10, ACM, 1995.

[4] Campbell, D. M., Embley, D. W., and Czejdo, B., "Graphical query formulation for an entity-relationship model," *Data & Knowledge Engineering*, vol. 2, no. 2, pp. 89–121, 1987.

[5] Chen, P. P., "An algebra for a directional binary entity-relationship model," in *Proceedings of the First International Conference on Data Engineering*, pp. 37–40, IEEE Computer Society, 1984.

[6] Chen, P. P.-S., "The entity-relationship modeltoward a unified view of data," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[7] Eisenberg, A. and Melton, J., "SQL/XML and the SQLX Informal Group of Companies," *Sigmod Record*, vol. 30, no. 3, pp. 105–108, 2001.

[8] Elmasri, R. and Navathe, S., *Fundamentals of Database Systems*. USA: Addison-Wesley Publishing Company, 7th ed., 2015 (forthcoming).

[9] Elmasri, R. and Wiederhold, G., "GORDAS: A Formal High-Level Query Language for the Entity-Relationship Model," in *Proceedings of the Second International Conference on the Entity-Relationship Approach to Information Modeling and Analysis*, ER '81, (Amsterdam, The Netherlands, The Netherlands), pp. 49–72, North-Holland Publishing Co., 1983.

[10] Färber, F., Cha, S. K., Primsch, J., Bornhövd, C., Sigg, S., and Lehner, W., "SAP HANA Database: Data Management for Modern Business Applications," *SIGMOD Rec.*, vol. 40, pp. 45–51, Jan. 2012.

[11] Gogolla, M. and Hohenstein, U., "Towards a semantic view of an extended entity-relationship model," *ACM Transactions on Database Systems (TODS)*, vol. 16, no. 3, pp. 369–416, 1991.

[12] Harizopoulos, S., Abadi, D. J., Madden, S., and Stonebraker, M., "OLTP Through the Looking Glass, and What We Found There," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 981–992, ACM, 2008.

[13] Harris, L. R., "User oriented data base query with the ROBOT natural language query system," *International Journal of Man-Machine Studies*, vol. 9, no. 6, pp. 697–713, 1977.

[14] Hohenstein, U. and Engels, G., "SQL/EER - Syntax and semantics of an entity-relationship-based query language," *Information Systems*, vol. 17, no. 3, pp. 209–242, 1992.

[15] Kemper, A. and Neumann, T., "HyPer: A hybrid OLTP & OLAP main memory database system based on virtual memory snapshots," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pp. 195–206, April 2011.

[16] Lawley, M. and Topor, R. W., "A Query Language for EER Schemas.," in *Australasian Database Conference*, pp. 292–304, 1994.

[17] Markowitz, V. M. and Raz, Y., "ERROL: an entity-relationship, role oriented, query language," in *ER*, pp. 329–345, 1983.

[18] Markowitz, V. M. and Raz, Y., "An entity-relationship algebra and its semantic description capabilities," *Journal of Systems and Software*, vol. 4, no. 2, pp. 147–162, 1984.

[19] Neumann, T., "Efficiently Compiling Efficient Query Plans for Modern Hardware," *Proc. VLDB Endow.*, vol. 4, pp. 539–550, June 2011.

[20] Neumann, T. and Leis, V., "Compiling database queries into machine code.," *IEEE Data Eng. Bull.*, vol. 37, no. 1, pp. 3–11, 2014.

[21] Omodeo, E. G. and Doberkat, E.-E., "Algebraic semantics of ER-models in the context of the calculus of relations: I: Static view," *Electronic Notes in Theoretical Computer Science*, vol. 44, no. 3, pp. 136–152, 2003.

[22] Parent, C., Rolin, H., Yetongnon, K., and Spaccapietra, S., "An ER Calculus for the Entity-Relationship Complex Model.," in *ER*, pp. 361–384, 1989.

[23] Parent, C. and Spaccapietra, S., "An algebra for a general entity-relationship model," *Software Engineering, IEEE Transactions on*, no. 7, pp. 634–643, 1985.

[24] Parent, C. and Spaccapietra, S., "A Model and an Algebra for Entity-Relation Type Database," *Technology and Science of Informatics*, vol. 6, no. 8, 1987.

[25] Roth, M. A., Korth, H. F., and Batory, D. S., "SQL/NF: A query language for ¬1NF relational databases," *Information Systems*, vol. 12, no. 1, pp. 99 – 114, 1987.

[26] Storey, V. C., "Relational database design based on the Entity-Relationship model," *Data & knowledge engineering*, vol. 7, no. 1, pp. 47–83, 1991.

[27] Tablan, V., Damljanovic, D., and Bontcheva, K., *A natural language query interface to structured information*. Springer, 2008.

[28] Wuu, G. T., "SERQL: An ER query language supporting temporal data retrieval," in *Computers and Communications, 1991. Conference Proceedings., Tenth Annual International Phoenix Conference on*, pp. 272–279, IEEE, 1991.